Istituto Italiano di Tecnologia
Advanced Robotics

Università di Genova

Model–Based Code Generation for the Kinematics and the Dynamics of Articulated Robots

Marco Frigerio

A thesis submitted for the degree of

Doctor of Philosophy (Ph.D.)

March 2013

Thesis supervisors:

Prof. Dr. Jonas Buchli
Agile & Dexterous Robotics Lab – Head
Institute of Robotics and Intelligent Systems
ETH Zurich

Dr. Claudio Semini
Dynamic Legged Systems Lab – Head
Advanced Robotics Department
Istituto Italiano di Tecnologia (IIT)

Prof. Dr. Darwin G. Caldwell
Advanced Robotics Department – Director
Istituto Italiano di Tecnologia (IIT)

a mio fratello Michele,
ai miei genitori,
e allo zio Carlo

Abstract

In the past decades of development of robotics technology, software has not received as much attention as control, mechanics and actuation. The goal of my thesis is therefore to contribute in reducing the gap between the current expertise in robotics (in fields like control theory, or multibody dynamics) and best practices of software engineering.
The thesis deals with software for articulated robots such as humanoids and quadrupeds. Recent trends in robotics research have focused on such machines, for applications that go beyond the use of manipulators in production lines. For example, legged robots capable of traversing rough terrain may be employed in the future to support human operators in the management of an environmental emergency, such as an earthquake.

The thesis initially describes the control system we realized for the quadruped robot HyQ, from the hardware platform to higher level software, and shows a selection of the promising results we have achieved. The thesis then illustrates the software models I designed about the kinematics and the dynamics of articulated robots in general, and the development of a code generation framework based on such models and on the technology of *Domain Specific Languages*. The code generation framework focuses on rigid body dynamics algorithms, such as the Newton–Euler algorithm for inverse dynamics, and on kinematics quantities like geometric Jacobians, which are crucial components of the control of articulated robots.

My approach allows the users to deal only with high level descriptions of a robot and relieves them from complex development of critical routines. Resources and efforts can then be focused on open research questions. The efficiency and speed of the generated code makes it suitable not only for simulations but also for real–time control of real robots. My framework exhibits flexibility and ease of use, and satisfies different requirements arising both from robotics, like real–time capability, and software engineering, like modularity and separation of concerns. Providing a toolchain that can effectively support research and development in robotics, based on sound software–engineering approaches, is the core contribution of my thesis.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Contribution . . . 4
- 1.3 Outline . . . 5

2 Basic concepts **7**
- 2.1 Software principles and features . . . 7
- 2.2 Error–feedback and model–based control . . . 9
- 2.3 Domain Specific Languages . . . 11
- 2.4 Overview of the code generation framework . . . 12

3 Related work **15**
- 3.1 Articulated robots . . . 15
- 3.2 Model–based control . . . 18
  - 3.2.1 Operational space control . . . 18
  - 3.2.2 Joint–level inverse dynamics control . . . 19
- 3.3 Rigid body dynamics . . . 20
- 3.4 Robotics software . . . 21
- 3.5 Robot modelling and code generation . . . 23

4 Software control of articulated robots **27**
- 4.1 Overview . . . 27
- 4.2 Key aspects of the platform . . . 30
  - 4.2.1 Real–time . . . 30
  - 4.2.2 Communication with the robot hardware . . . 32
  - 4.2.3 Motor control and robot behavior . . . 39
  - 4.2.4 The SL package . . . 41

5 Domain analysis and software models **43**
- 5.1 Kinematics and dynamics of articulated robots . . . 43
  - 5.1.1 Kinematics . . . 44
  - 5.1.2 Dynamics . . . 46
- 5.2 Domain models . . . 48
  - 5.2.1 Kinematic trees . . . 49
  - 5.2.2 Rigid body motions . . . 55

6 The code generation framework **59**
6.1 The specification languages 59
6.1.1 Overview 60
6.1.2 General features of the grammars 60
6.1.3 The kinematics DSL 60
6.1.4 The rigid body motions DSL 65
6.1.5 The coordinate transforms DSL 65
6.2 Code generation 68
6.2.1 Robot–specific dynamics routines 68
6.2.2 Coordinate transforms 70
6.2.3 Putting all together: the robotics code generator 75

7 Experimental results **79**
7.1 Control software for articulated robots 79
7.2 Evaluation of the code generator 83
7.2.1 General remarks 83
7.2.2 Validation 84
7.2.3 Performance comparisons 84
7.2.4 Simulation and control 88

8 Conclusions and future work **91**
8.1 A software architecture for the HyQ robot 92
8.2 The robotics code generator 94
8.2.1 Discussion 94
8.2.2 Future improvements 96

A Generated code example **99**
A.1 Header file 99
A.2 Definitions file 101

B Publications **105**

List of Figures

2.1 Model–based control vs. error–feddback control 10
2.2 Overview of the code generation approach 13

3.1 The HyQ robot 16
3.2 Articulated robots 17

4.1 The computing setup of the HyQ robot 28
4.2 The computer and the I/O boards of HyQ 29
4.3 Simplified architecture of Xenomai 32
4.4 The hardware–software boundary 34
4.5 The Sensoray526 software stack 35
4.6 The class diagram for encoders 36
4.7 A generic control scheme 40
4.8 The graphical interface of the SL software 42

5.1 Ambiguity in rotation matrices 45
5.2 A simple example of a geometric Jacobian 46
5.3 Modelling a robot as a graph 50
5.4 Kinematic trees 51
5.5 Layout of reference frames for a generic section of a kinematic chain 52
5.6 The kinematic tree UML model 54
5.7 The UML class diagram for rigid motions 55
5.8 The UML class diagram for coordinate transforms 56
5.9 Adding Jacobians in the UML class diagram 58

6.1 The relation between language grammars and domain models . . 61
6.2 The grammar of the kinematics DSL 62
6.3 The HyQ robot model in a document of the kinematics DSL . . . 64
6.4 The grammar of the motion DSL 66
6.5 A document of the motion DSL 66
6.6 The grammar of the transforms DSL 67
6.7 A document of the transforms DSL 67
6.8 Robot specific implementations 69
6.9 A document of the DSL for MAXIMA 73
6.10 Code generation for coordinate transforms 74
6.11 Specifying the desired transforms 75
6.12 The code generation workflow 76
6.13 From joint parameters to a motion DSL document 76

7.1 Walking experiments with the HyQ robot 80
7.2 The C++ interface for hydraulic valves 80
7.3 Hopping experiment with the hydraulic leg 81
7.4 Force tracking during squat jump 82
7.5 The HyQ robot performing more advanced tasks 82
7.6 Comparison with SL of the speed of dynamics algorithms 85
7.7 Comparison of the computation time of the JSIM 86
7.8 Comparison of the computation time of the null space projector . 87
7.9 Simulation with the generated code 88
7.10 The generated code for inverse dynamics running on a real robot 89

Chapter 1

Introduction

Designing software for robots is among the most demanding and complex software engineering challenges, due to a list of strict and partially conflicting requirements. The sheer complexity arises from the high number of tasks such a software has to perform, the variety of characteristics of the different modules (e.g. computational complexity, frequency of execution, etc.) and the need of effective orchestration.
For the robotics research community as well as for a widespread adoption of robotic technology, it is central to have flexible yet reliable software: a typical academic research unit cannot afford the same resources to develop reliable software as an airplane or a car manufacturer, yet requires safe and flexible solutions for a machinery of similar complexity, in order to address open research questions.

However, software is often conceived as a mere tool to achieve the numerical implementation of some control scheme, in order to get some visible results as quickly as possible. On the one hand, this approach is justified and actually required, each time one addresses new research problems for which an established, general solution does not exist. On the other hand, it is of fundamental importance to realize that software itself represents a critical part of robotics development as much as mechanics and control. A careful design, and an appropriate choice of implementation technologies are simply necessary (not sufficient) conditions to achieve useful and versatile robots. In general, this work can be classified within the research field whose main purpose is to fill the gap between software engineering practices and robotics software development.

This thesis describes the first computing system we developed for the HyQ quadruped robot, which allowed us to perform a variety of experiments to investigate artificial legged locomotion. This experience and evidence in the literature showed the importance of approaches based on physics models (e.g. multibody dynamics) for high performance control of articulated robots; hence, the second part of the thesis focuses on an approach to provide robust and efficient implementation of kinematics and dynamics computations for such a class of robots.

1.1 Motivation

Almost any software application for articulated robots needs to perform some calculus related to the geometry of the robot. Control approaches based on a sound physical model of the robot (a multibody system) enable more sophisticated behaviors by taking advantage of the high degree of mobility of these machines. Examples of model–based controllers include operational space controllers (Khatib 1987) and impedance controllers (Hogan 1985), which heavily rely on kinematics and dynamics algorithms and expressions.

The algorithms for kinematics and dynamics are well known in the robotics community and have been extensively studied in the past decades. However, despite the established theoretical understanding of them, sound implementations demand a lot of resources and still represent an obstacle for the development of new applications: a significant development in the starting phase of a project is required to make the robot operational, it is critical for the control and simulation but often not the focus of the research per se (for example if a researcher wants to test some learning algorithms on a new manipulator).

The difficulties arise from the lack of established software *models* and thus the lack of *reusable* components, addressing the inherent complexity of kinematics/dynamics (the availability of several textbooks on this subject does not imply they are trivial). The scarce availability of truly reusable solutions, often results in wasting significant development time on what we can call *non–problems*. These problems have been solved theoretically and covered in the literature, nonetheless practical solutions like software implementations can still be hard to achieve, with the unfortunate consequence of turning such non–problems back to real ones. Coordinate transformation matrices are a simple, yet meaningful example of a recurring non–problem of almost any robotics software application. Such matrices are conceptually relatively simple and they are considered text–book knowledge. However anyone who has ever worked with these objects is aware of how confusing they can be, and how much effort is required to validate any implementation using them.

Writing software becomes even more challenging when real–time constraints and limited hardware resources demand fast and efficient code. The real–time capability is fundamental for the implementation of control algorithms running on real robot hardware; any failure of the software may lead to consequences such as physical damage to the robot itself or the surroundings, including people. Code used for model–based controllers is a typical example of software with an apparent trade-off between flexibility/maintainability and efficiency. On one hand a rigid body model is a generic description of a robot that naturally lends itself for a rather general implementation, e.g. with object oriented code. On the other hand it is critical that such code does not violate real–time constraints (e.g. by system calls such as those for dynamic memory allocation or file access) and is ideally running as fast as possible (e.g. exhaustive evaluations in sampling based planner algorithms or fast control loops). Resolving such a trade-off would substantially improve the quality of software without compromising its effectiveness for robotics applications.

In the past (say 15-20 years ago, or more), the need for optimized code in robotics was perceived as a more critical issue than today, since the same requirements of the real–time control of robots had to be met by much slower

computers. Nowadays, computing power is much higher and cheaper, nonetheless some of the issues of hard real–time code for robot control are still retained. First of all, the specific requirements of real–time capable code have not changed: special care must be taken not to insert system calls like input/output operations or similar, which have a non–deterministic execution time (often also "long", relative to the timing constants of the control loop) and therefore may violate the strict deadlines of the control loop.

In addition, in parallel to the evolution of computing resources, achievements in robotics and the expectation of the community about robotics have grown. Therefore, barely managing to run a PID position controller implemented on a separate workstation connected to the robot is not satisfactory. Research is striving for fully autonomous robots, i.e. machines embedding all the required power and the intelligence to operate in unstructured environments; in addition, despite the evolution of hardware, embedded computers typically mounted on a robot may still have some limitations in terms of memory, CPU, etc. Therefore, the faster the implementation of certain routines (e.g. kinematics and dynamics), the larger the room for additional computation within the same hardware resource boundaries. More sophisticated logic leading to more effective behaviors of the robot can then be implemented.

In addition – regardless of the available computing power – there are always certain tasks whose implementation is ideally as fast as possible, since higher speeds would enable more effective results. The simulation of the dynamics of a multibody system is a typical example of this idea: the faster the simulation, the more convenient it is for the user to try out new approaches, use more complex models, etc. Another interesting scenario would be when the simulation is executed on–line on a robot moving in the environment, e.g. within a sampling–based planning routine: in essence, the robot can use its own dynamic model to evaluate different alternative movements it might take as a reaction to a certain event (e.g. a slippage, a shift of its current target, etc.). The faster the robot can simulate possible scenarios, the more reactive and therefore versatile it can be, since it can adapt to sudden events.

For all these reasons, roboticists would benefit from an automatic implementation of kinematics and dynamics computations, in terms of robustness of the code and resources saved during the course of the project. Dynamics algorithms, for instance, are general and parametrized on the kinematic description of a robot – often called the robot *model* (Featherstone 2008) – which is relatively compact but fully specifies the physics of the system. Hence, it is sensible to look for a high level representation of the robot models, which can be easily constructed by hand, while exploiting automated procedures to turn such information into code. See Section 5.1.2.3 and 6.2.1.

A general dynamics library (such as ODE (Smith 2013)) – which would necessarily require the robot model as a *parameter* – could solve this problem. But the main point of generating code is efficiency without loosing flexibility. In any case, a robot is unlikely to need a general purpose dynamics engine that can solve a whole *class* of problems, when it only needs to solve the dynamics of its own body.[1] To keep the software lightweight and efficient, it is more effective

[1] Very advanced control software computing the dynamics of other mechanisms the robot may interact with – such as a door – is an exception of this point.

to have a specialized implementation.

1.2 Contribution

To address the points described above, this thesis is focused on a toolchain for the automatic code generation of algorithms for rigid body dynamics and kinematics, for the simulation *and* the control of robots under real–time constraints. Our focus is on articulated robots, made of chains or branched chains of rigid links.

The code generation framework is based on a few Domain Specific Languages (Section 2.3) concerning kinematics and dynamics, whose design is based in turn on general domain models. The resulting system represents a progress beyond state-of-the-art tools since it gathers different effective features – usually not available together. In particular, the main contributions of my work are summarized in the following points:

- The design of simple yet general domain models about the kinematics and the dynamics of articulated robots, according to an established best practice in software design (in a field, robotics, that often partially lack a structured development process (Bischoff, Guhl, et al. 2010)). These models enable a principled development of the toolchain and also ease the understanding of the end users (Section 5.2).
- The exploitation of Domain Specific Languages, as an effective technology for the implementation of our approach. A sound choice of the technology again helps the development as well as the user experience (Section 6.1). For example, users can write the description of the kinematics of their robot with a simple text file that has a very intuitive format. If necessary, the file can be converted to other, less user–friendly formats automatically.
- The development of a code generation framework, based on the previous points, which exhibits several desirable features:
  - Ease of use: the user is required to deal only with high level information and is relieved from time consuming and error prone code development.
  - Reliability and robustness: an automated code generation process is repeatable and cannot introduce occasional mistakes, as opposed to manual coding.
  - Efficiency: the generated implementations can be optimized to address the speed and efficiency constraints of real–time robot controllers.
  - Flexibility: domain models make the software more general and enable extensibility. Any articulated robot (without loops) is supported, arbitrary coordinate transforms can be chosen, etc.
- The achievement of a software that does not sacrifice performance for generality or the other way round. For instance, the generated C++ code is suitable for hard real–time control of real robots, yet the framework is easy to use and works for a wide class of robots.

- An implementation of a model–based code generation approach based on open source technologies only. This property makes it very convenient for the user to install and use the toolchain, easing a widespread adoption of this work.

An additional contribution lies in the development of a general software system for the control of articulated robots, which allows users to robustly operate with real machines. This system, for example, allows the roboticist to perform experiments and thus address state-of-the-art research questions.

1.3 Outline

The rest of this thesis is organized as follows:

- Chapter 2 serves as an introduction to different topics and concepts related to software and control, required to understand the rest of the material. This chapter also contains a general overview of the code generation framework, in Section 2.4.
- Chapter 3 discusses several works in the research literature concerning the topics discussed in this thesis – e.g. model based control, software for robotics.
- Chapter 4 describes the control system for the HyQ robot, as a representative solution for the control of a sophisticated articulated robot.
- Chapter 5 and 6 illustrate the domain analysis and the design we realized to implement our robotics code generator. These two chapters – together with the previous one – contain the core material of the thesis.
- Chapter 7 presents an overview of the promising results we achieved with the HyQ robot, thanks to its control software. The chapter also shows some experimental results with our code generation framework, e.g. in terms of performance comparisons.
- Finally, Chapter 8 discusses some of the current issues and proposes directions for future development.

Chapter 2

Basic concepts

The goal of this chapter is to introduce some of the basic concepts that will be used throughout the rest of the text. The material in this chapter is a prerequisite for a proper understanding of the following ones.

Section 2.1 lists some general principles and best practices about software engineering that happen to be particularly relevant for robotics software. Section 2.2 gives an intuitive explanation of model–based control techniques, while Section 2.3 introduces the technology of *Domain Specific Languages*.

2.1 Software principles and features

This section lists some very general desirable features for software, which often are not specific to robotics but are nonetheless particularly relevant for this field. The concepts described in the following points will be referenced in the rest of this thesis.

Domain models Appropriate abstractions for common components, and in general for *recurring problems* in robotics, fosters the establishment of principled and general solutions (e.g. a reference implementation in the C language of a PID controller or a general model of virtual components for operational space control (Pratt et al. 2001)). Although this practice is well known in software engineering in general, the development of robotics software often does not follow a rigorous approach (Bischoff, Guhl, et al. 2010); developers often "reinvent the wheel".

Separation of concerns This is another general principle – mentioned for instance in chapter 4 –, which refers to the proper partitioning of the tasks of the system and of the corresponding implementation. For example, it is important to impose a clear separation between the low–level modules for motor control and those pertaining the behavior of the whole robot. *Modularity* is strictly related to *reusability*, a property that also alleviates the reinvent–the–wheel issue. Note that reusability does not apply only to code, but also to models, design patterns, strategies, etc.

Explicit properties and semantics The lack of proper information on underlying models, assumptions, and even the function itself of software

components, poses severe limitations on the reusability of such components. System integration costs rise, sometimes up to the point when it appears to be more convenient to develop components from scratch. This problem is quite severe in robotics since the composite nature of software dealing with a diversity of fields (e.g. control theory, image processing, kinematics) makes the integration a critical point. In addition, specific fields such as rigid body kinematics may easily lead to ambiguous implementations because of non–standard conventions or implicit assumptions (Laet, Bellens, Smits, et al. 2012). Note that the simplest approaches to address this issue are simply documentation and possibly standardization; such remedies are at the user level. Sometimes though it is necessary or desirable to explicitly represent certain semantics (e.g. by adding more properties to a data type) to enable a more automated integration (e.g. a software that composes correctly a sequence of coordinate transformation matrices).

Other points, somehow more directly related to the actual implementation, include the following:

Targeting different platforms The development and the experimentation with robots may require diverse hardware platforms and technologies. Maybe the most notable example of this point is the need of working both with simulators and with real machines. Simulation is necessary since experimentation of new approaches (that may fail) on a real robot is typically much more demanding in terms of resources; these costs, which however vary heavily with the type of the robot (big, small, powerful, articulated, etc.), include the time to check and activate the platform, the number of involved people, all the costs related to a possible damage to the robot itself.
Therefore, it is desirable to have support for multi–platform development, for example to ensure consistency between the C code to be deployed on the real robot and the MATLAB code reserved for simulations.

Reliability and robustness We may describe these two properties of a human–made system respectively as the likelihood and the confidence about a proper functioning and the capability of dealing with a diversity of operating conditions; robustness also refers to how gracefully the system fails as a consequence of non–manageable events (*exceptions*). Obviously, these properties are relevant for any kind of software, but at different degrees that depend on the criticality of the application. The possibly high maintenance costs associated to failures of a robot, but especially the safety issues arising in the field of human–robot interaction make robotics a domain where guarantees about reliability and robustness are very important. The more powerful the actuation of the robot, the more critical the safety issue, since the robot is potentially more harmful.

Debugging support The complexity of a typical software platform for a robot and the need for reliability (discussed above) demand for effective tools for testing and debugging. Inspection of the status of the software (e.g. the current value of some variable) is necessary also while performing actual experiments. Data loggers (possibly compatible with real–time pro-

cesses), graphic interfaces, simulation tools are precious resources to evaluate the correctness of the implementation and to identify the source of problems during an experiment. Simulators can help in estimating the performance of control strategies, graphic visualization can give the user a quick overview of part of the data (e.g. the layout of the reference frames on the bodies of the robot) providing immediate clues of malfunctioning (e.g. a broken position sensor results in a weird posture of the robot model displayed on a screen) and logs can be inspected off–line to spot the source of numerical problems. For some of these features there exist generic implementations that can be applied to many cases; for example in ROS (Quigley et al. 2009) there is a generic component which can visualize a graphical model of the robot, called RViz.

Real–time and efficiency This topic will be discussed in Chapter 4, especially in Section 4.2.1, but it is briefly introduced here for convenience. A critical requirement for some of the code of the software of a robot is to be compatible with real–time constraints, since it must execute within a certain deadline in order to lead to correct results. Entire portions of code have to be developed and maintained with specific care so as not to use incompatible operations with unpredictable execution time.
Due to space or power constraints, robots often mount embedded computers that might provide limited hardware resources and thus run–time capabilities. For this reason, the efficiency of the software is also very important and might impose additional constraints on the implementation, like prohibiting the use of a certain library. Tools capable of assessing memory and computational complexity of parts of the code can help the development.
In addition, although efficiency (and speed) are in principle orthogonal to real–time capability, in robotics they happen to be tightly related because real–time routines usually have to be very *fast* due to specific requirements (e.g. control loops running at hundreds of Hertz). Hence, it is even harder to develop by hand code suitable for certain software components. The choice of implementation and deployment technologies is also restricted because of the need for speed guarantees: for example, certain programming languages cannot be used, regular off–the–shelf communication protocols might be too slow, etc.

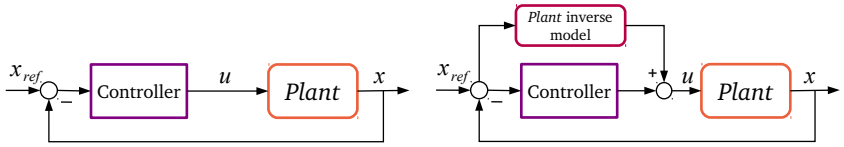
2.2 Error–feedback and model–based control

Generally speaking, control theory gives us methods to drive the behavior of a real system (often called *plant* – e.g. an assembly of rigid bodies) by means of numerical techniques, i.e. algorithms that can be executed on computers. These algorithms are typically based on a periodic sequence of operations that involve injecting some sort of input in the plant, based on the measurement of state and output variables of the plant itself.
As an example, it is possible to move each joint of a robot up to a certain desired position, by activating the actuator while constantly monitoring the current position (status) of the joints.

A simple form of automatic control of a plant is the *error feedback* control,

that is a strategy purely based on the comparison between the expected and the actual value of the quantity to be controlled. The actual logic of the control is thus very simple and boils down to a weighting of such an error, possibly using also the derivative and the integral of the error itself. It follows that such a control approach is basically agnostic with respect to the actual plant to be controlled, since it does not rely on any particular assumption; however, it is true that the weights of the control function (called *gains* in the literature) have to be chosen appropriately in respect to the plant dynamics.

As a matter of fact, there exist more sophisticated control strategies that exploit some knowledge about the plant, and can be used in combination with other strategies (a detailed discussion about control architectures is beyond the scope of this work). The term *model–based* control refers to such a class of strategies that rely on some sort of approximate, formal description of the internal behavior of the plant (i.e. a model). Figure 2.1 illustrates this concept



(a) Purely error feedback **(b)** Model based

Figure 2.1 – A minimal illustration of two different control strategies: in the first approach the only input to the controller is the difference between the outcome of the plant and the desired reference point. In the second, some knowledge about the plant internals is exploited to estimate the required input in a principled way; note that the error feedback branch is still active, to cope with the uncertainties in the plant model.

through a simple comparison of two generic block diagrams, representing a control loop based on the feedback of the error and one using a model of the plant.

The rationale behind the use of model–based approaches is quite intuitive: using some prior information about the plant in the controller should lead in general to better or more effective results (e.g. faster responses) in comparison to a purely generic approach (such as a PID controller).

For articulated robots such as humanoids and other legged robots, the model being exploited in control is typically that of *Rigid Body Dynamics* (RBD). We do not use information strictly specific to a certain robot but rather rely on our knowledge about the physics of rigid bodies in general, i.e. the relation between motion, forces, and inertia. We use the term "model" since the whole theory is still based on an simplified view of the reality: it is assumed for all inertial bodies to be perfectly rigid, which is not true in general but it is often a good approximation. Moreover, RBD deals only with a basic set of physical quantities, namely velocity, force and inertia, and essentially it describes the effect of forces on the motion of bodies (i.e. dynamics). This is one of the reasons why it is desirable to have a fully force–controlled robot, as shown for instance in Figure 4.7: it allows us to abstract the details of the actuation system and to think of it as a force source, thus enabling the use of RBD based approaches.

Each articulated robot is treated as an assembly of many rigid bodies (often called "links"), with force sources at the joints (a *multibody* system); more on the modelling of articulated robots will be presented in Section 5.2.1. Model–based control has proved to be a fundamental approach to achieve more sophisticated behaviors and to increase the performance of articulated robots. Refer for instance to (Boaventura, Semini, Buchli, Frigerio, et al. 2012; Focchi et al. 2012; Mistry, Buchli, et al. 2010; Sentis et al. 2005).
Maybe the most common example of a model–based controller is the inverse-dynamics–based controller (see Section 5.1.2.2), which uses the dynamic model of the robot to estimate in advance the forces required to achieve the desired accelerations. With this approach it is possible to achieve very fast movements of the links of the robot that a regular PID controller cannot track well: i.e. the actual movement does not reproduce with good accuracy the desired trajectory (Boaventura, Semini, Buchli, Frigerio, et al. 2012). In general, this approach allows the reduction of the contribution of the error feedback control (that can run in parallel) to the final commands delivered to the actuators, since it pre–computes a good estimate of the necessary forces. The gains of the error feedback function can then be reduced, resulting in a more compliant behavior of the joints of the robot, which can be a very useful feature for certain tasks such as locomotion.

2.3 Domain Specific Languages

This sections gives a brief introduction to the idea of Domain Specific Languages (DSLs). The aim of such an introduction is to give a general, reasonable understanding of the topic, focusing on the features that are most relevant with respect to the content of the next chapters; a detailed explanation would go beyond the scope of this work, and the interested reader can refer to publications available in the literature (Fowler 2010). How DSLs fit into our approach is detailed in Section 6.1.

A Domain Specific Language is a computer language suitable to provide some sort of specification related to a precise class of problems (i.e. a domain) (Mernik et al. 2005). The syntax and especially the semantic of the language are explicitly designed to have a limited expressiveness in general, which is paid off by the clarity in the representation of the elements of the target domain. The level of abstraction of the documents is then higher than with a regular programming language, making DSLs more suitable also for non–developers and domain experts.
Domain specific languages naturally compare against the so–called general purpose languages, which on the other hand are programming languages with lots of capabilities and that can implement any algorithm. However, the concept of DSL does not have sharp boundaries, and the classification of certain languages can sometimes be argued.
Many examples of DSLs exist in the world of computer programming, and we can cite a few: SQL to specify queries to databases, regular expressions to specify character patterns used for text manipulation, custom scripts of any application to specify a batch of operations to be performed.

DSLs can be roughly divided into two categories: "internal" and "external".

Internal DSLs include languages built through a particular usage of an existing general purpose language, using its syntax and a subset of the language features. External DSLs include independent languages that usually have a custom syntax (Fowler 2010). Confirming the sometimes blurred boundaries of these definitions, some examples are harder to classify; think for instance of XML based languages, which can be considered external DSLs using the syntax of an existing language. The URDF file format used in ROS to describe the robot kinematics is an example of this case (Quigley et al. 2009).

For our work, it is very useful to think of a DSL as a thin layer on top of a domain model that describes the actual structure of the information and thus most of the semantics of the language itself. The language becomes simply a convenient yet clear tool for the specification of instances of the model, by means of documents compliant with the language (Fowler 2010).[1]
The last important point about DSLs concerns run–time execution, that is, how they concretely contribute to the operations of a software system. We are interested here in code generation, which is basically a transformation process of the DSL documents into source code of some programming language that can be compiled and executed. Code generators use the information encoded in the documents during the generation of code, according to some logic (e.g. an algorithm to be implemented with some parameters taken from the document, a predefined template to be filled with values, etc.).

2.4 Overview of the code generation framework

The purpose of this section is to give the reader an overview of our code generation framework for the kinematics and the dynamics of articulated robots. The domain analysis and the implementation of the parts of the generator will be detailed in Chapter 5 and 6 (pointers to specific sections of these chapters will be given below). The aim of this section is to describe the conceptual framework such parts fit into, to prepare the reader to a more technical explanation.

This introduction – together with the points listed in Section 1.2 – also provides a first evidence about how the framework meets some of the software principles listed above (Section 2.1), which are particularly significant for robotics.

Figure 2.2 illustrates the idea behind the code generation approach. The core logic resides in the block called DSLs *infrastructure*, which includes the grammar and the parser of our DSLs, plus the code generation algorithms that transform parsed documents into code. The input information for this block is conveyed by *high–level models* in the form of documents of the languages; for example, these documents can contain a description of the kinematics of a robot, or the list of translations and rotations that relate two different reference frames of interest. Such high–level information is the only thing the final user has to deal with.
Note that the code generation software embeds the full knowledge about the specific algorithms whose implementation is of interest, as for instance the *Recursive*

[1] Actually Fowler uses the term "semantic model", to mean a part of the whole domain model, and identifies each DSL document with *a* semantic model, rather than talking about instances.

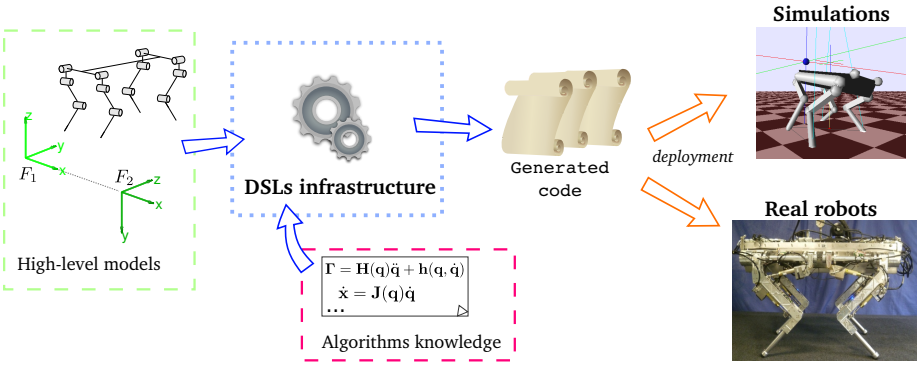


Figure 2.2 – A high level overview of the code generation framework. The core logic resides in the the DSLs infrastructure, which includes code generation software embedding the knowledge about kinematics and dynamics algorithms. The actual input to the framework are high level models typically in the form of documents of some DSL. The output is standalone code in different languages, which can be used both for simulations and control of real robots.

Newton–Euler algorithm for inverse dynamics (see Section 5.1.2.2). Hence the algorithm itself is currently not a specific input to the infrastructure, but rather a piece of information embedded into it.

The infrastructure generates source code possibly in different programming languages, and possibly tailored for specific platforms. The code generation process can also perform various optimizations making the output code quite efficient. For example, as explained in Section 5.1.2.3, dynamics algorithms are parametrized with respect to the robot structure: knowing such structure allows the generation of tailored implementations that are more efficient, that is, specific *instances* of the general algorithm. An instance of an algorithm is a particular realization of the algorithm itself obtained by fixing one or more of the parameters. Basically each parametrized algorithm can be considered as a *class* of procedures, each one identified by a specific value of some of the parameters. See also Section 6.2.1.

The generator produces the implementation of functional components (such as a dynamics algorithm), which have a precise, well identified meaning and can be reused in different external applications. In addition, our approach allows the users to deal only with high level information and relieves them from problematic hand-crafted development; resources and efforts can then be focused on open research questions.
The flexibility of the framework as far as languages, platforms and efficiency are concerned, makes it suitable for simulations but also for the control of real robots under hard real–time constraints. A single generation infrastructure, and a single format for input information guarantee the consistency among the different outputs; this property is fundamental e.g. when working with both the simulation and the control of the same robot on different platforms.

Two of the DSLs the framework is comprised of are actually standalone tools whose aim is to provide a toolchain to perform robust code generation of ar-

bitrary coordinate transforms (see Section 6.1.4 and 6.1.5); it is definitely not effective to develop transforms by hand, besides for learning purposes, and any roboticist should not waste time with this non–problem. How these languages fit into the bigger framework of code generation for articulated robots is detailed in Section 6.2.3.

Currently, the following quantities and algorithms are supported by the code generation framework (the target languages are mainly C++ and MATLAB, but more may be added):

- The *Recursive Newton–Euler algorithm* to compute inverse dynamics, the *Articulated–Body algorithm* for forward dynamics, and the *Composite–Rigid–Body algorithm* to compute the joint–space inertia matrix. See Section 5.1.2.2 and 6.2.1.
- Arbitrary coordinate transforms (for homogeneous coordinates and for spatial vectors) and arbitrary geometric Jacobians, for any point of interest on the robot structure. See Section 5.1.1.1, 5.1.1.2, 6.2.2 and 6.2.3.1.
- A few robot–description files can also be generated by our kinematics DSL (see Section 5.2.1, 6.1.3), as for instance the URDF-XML file used in ROS (Quigley et al. 2009), and the text file required by SD/FAST (Sherman et al. 2013).

Chapter 3

Related work

3.1 Articulated robots

Articulated robots are machines composed of multiple rigid bodies connected via joints that allow some degrees of motion freedom. Actuators provide the energy required to *articulate* the joints and therefore move the links. Articulated robots range from linear chains of bodies connected in a sequence, such as industrial manipulators or snake–like robots, to more complicated branched structures such as humanoid robots, typically with more degrees of freedom (DoFs).
Articulated robots represent a wide class of machines largely used in industry and research, which differ from simpler mobile platforms (e.g. wheeled rovers equipped with sensors) because of the larger number of DoFs and thus the greater dexterity. Think for instance of the manipulation capability of a human arm, or the degree of mobility of mammals with four legs.

HyQ (Figure 3.1) is a versatile quadruped robot developed at the Istituto Italiano di Tecnologia in my research group, and it is a notable example of an articulated robot. HyQ is about one meter long and 70 kg heavy, with electric and hydraulic actuation at its twelve joints, three for each one of the four legs. It is primarily designed to investigate artificial legged locomotion, which holds the promise of building machines able to traverse very rough terrains precluded to wheels and tracks (Semini, Buchli, et al. 2011; Semini, Tsagarakis, et al. 2011). The actual purpose of the project is to make the robot capable of complex behaviors ranging from highly dynamic tasks, such as running or jumping, to slow and careful walking over rough terrain, when mapping and planning become necessary. A challenging objective is to make the robot fully autonomous with respect to both the energy consumption and the control.
Hydraulic actuation was chosen primarily because of the speed and the power, but also because of the capability of the actuators (like cylinders) to withstand strong impacts, inevitable in legged locomotion. A high–performance, full–body active torque control is the strategy being investigated in this research project to enable the versatility required for locomotion.
Chapter 4 describes the computing system currently installed in HyQ that enables the control of the machine through software programs.

(Hutter et al. 2012) presents another quadruped robot – called StarlETH – designed for similar purposes, but smaller in size (about 0.6 meters in length

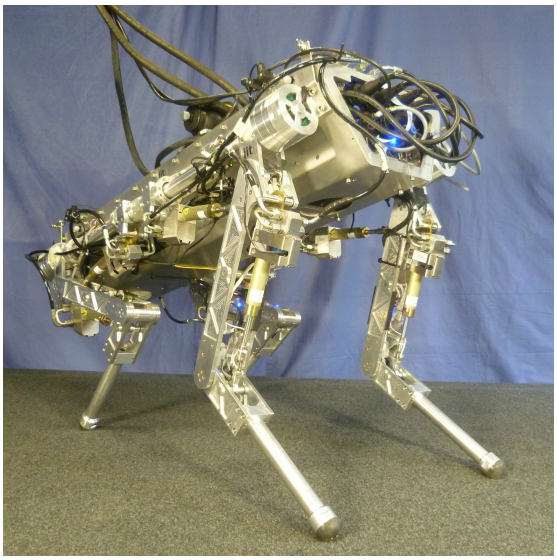


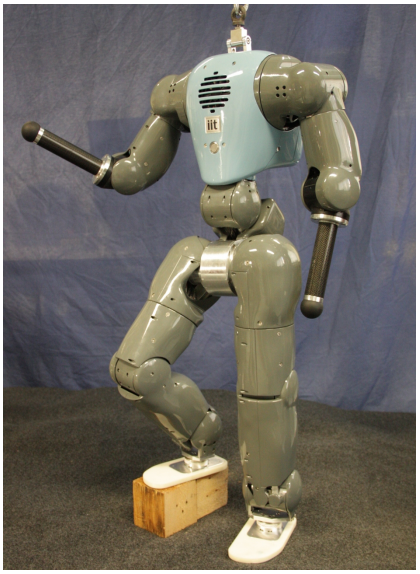
Figure 3.1 – The hydraulically actuated quadruped robot HyQ, developed at the Advanced Robotics Department of the Istituto Italiano di Tecnologia.

and 23 kg weight) and actuated only by electric motors with elastic elements in the transmission. The development is focused on the compliance of the series elastic actuation as the key to cope with collisions with the ground and to enable safe interaction with humans. Another objective of such an actuation system is to increase the overall efficiency of the robot.

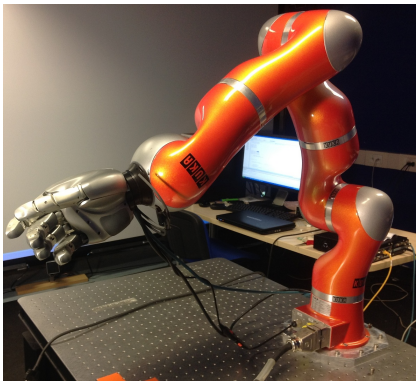
The COMpliant huMANoid robot COMAN is a full humanoid robot, approximately the size of a four year old child, which weighs about 31 kg and has a total of 25 DoFs (Tsagarakis et al. 2013). COMAN was also designed and built with passive compliance actuators in some joints, to cope with impacts and to be more adaptable during the interaction with the environment. The investigation of how passive compliance can improve the performance of such a robot is an explicit goal of the research. The paper illustrates the mechanical design of the robot, focusing on the *series elastic actuators* module for the joints, the legs and the torso of the humanoid. Before showing some experimental trials, the authors also describe the dynamics model of the robot and their strategy to optimally determine stiffness values for the 14 joints with compliance.

The Lightweight Robot (LWR) is an articulated robotic arm jointly developed at KUKA Roboter and the Institute of Robotics and Mechatronics at the German Aerospace Center(DLR) (Bischoff, Kurth, et al. 2010). This robot was designed to achieve high dynamic performance and to be suitable for the current trend of robotics research, the human-robot interaction: limited mass for lower power consumption and easier control, seven degrees of freedom for high mobility, torque sensors at each joint and fast control loops to enable compliant behavior and model–based control schemes.

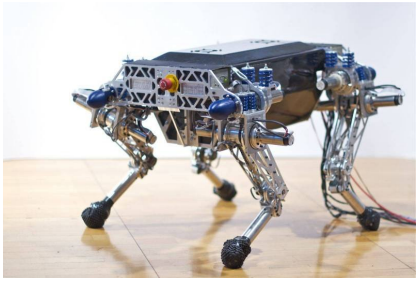
Pictures of the robots described in these paragraphs can be found in Figure 3.2.



(a) The COMpliant huMANoid robot COMAN (Tsagarakis et al. 2013)



(b) The KUKA–DLR lightweight robot arm (Bischoff, Kurth, et al. 2010)



(c) The StarlETH quadruped (Hutter et al. 2012)

Figure 3.2 – A few examples of articulated robots currently being developed in research laboratories.

3.2 Model–based control

The potentially high mobility and dexterity of articulated robots such as those mentioned above, and the development of new trends in robotics applications that go beyond the use of manipulators in industrial production lines (e.g. complex manipulation, interaction with humans) demand for more sophisticated control schemes compared to traditional joint–position control.

3.2.1 Operational space control

The early work of Khatib is generally considered as the foundation of the operational space formulation, an approach shifting the focus of control from the single joints of the robot to the actual *task*, i.e. the behavior of one or more points on the robot (typically the end–effectors). In (Khatib 1987), the author describes the modeling of the dynamic behavior of the end–effector of a manipulator (as opposed to the dynamic model at the joint level), to devise a control framework addressing both the position and the contact forces at the end–effector. The development of the approach stems from the basic idea that the allowed motions of a constrained end–effector (like a tool pressing on a flat surface) form a certain vector space, while the forces the end–effector can exert lie in another space which is orthogonal to the first (i.e. the tool can press but not move along the normal to the surface, and it can move in the other directions; constraint forces do not do any work)[1]. In (Khatib 1987), the dynamics equations relating the end–effector with the joints are developed, and then the treatment is extended to the case of redundant manipulators (i.e. those having more degrees of motion freedom in the kinematic structure than the degrees of freedom of the current task – e.g. a 7-DoF arm with respect to the positioning of the hand in space, which is a 6-DoF task). These systems are special because the relation between forces in joint space and task space is not bijective. Note that the operational space dynamics is basically the result of *projecting* the joint space dynamics into the more convenient motion space of the end–effector. In other words it is a different description of the same physical system, which can be more convenient for the formulation of controllers.
In (Khatib 1995) the operational space formulation is revised and new results are shown, with reference to macro-mini manipulators and parallel multiarm structures. This work summarizes the dynamics equations, establishes some nomenclature (like *basic Jacobian*, and *dynamic consistency*) and is the main reference for work in this field.

A development of the operational space formulation that started in more recent years and has experienced a growing interest from the community, pertains whole body control of branched robots such as humanoids and the hierarchical decomposition of multiple tasks (Khatib et al. 2004; Sentis et al. 2005).
This field or research is mainly motivated by the new emerging applications of humanoid robots, interacting heavily with the environment and with human operators. In (Khatib et al. 2004) and (Sentis et al. 2005), for instance, it is

[1] Note that, in this case, the concept of orthogonality is based on the scalar product defined between velocities and forces, which live in two different non-Euclidean vector spaces. So it is not the same thing as the common notion of orthogonality defined within an Euclidean space (such as $\mathbb{R}^3$) where an *inner* scalar product between vectors of the same space is defined (Featherstone 2008; Siciliano et al. 2009)

argued that the human motion is the result of the simultaneous execution of multiple tasks with different priorities, such as balancing, walking, manipulating objects, suggesting a control approach that mirrors this structure.
Multiple tasks each identifying an operational space are then defined and organized into a prioritized hierarchy where, for example, the balancing of the humanoid is the most important task. The concept of dynamic consistency first introduced in (Khatib 1987) is applied to ensure that control actions aimed at fulfilling a certain task are guaranteed not to affect higher priority tasks. The fundamental tool in the algebra of the approach is in fact the task null–space projector, a matrix mapping a torque vector into the closest vector that does not lead to accelerations violating certain constraints. Quite intuitively, a precondition for the applicability of the approach is some degree of redundancy in the robot, which is typically available in humanoids.

3.2.2 Joint–level inverse dynamics control

Model based control can be very effective also at the joint level, as in the case of controllers based on inverse dynamics (see Section 2.2).
In (Boaventura, Semini, Buchli, Frigerio, et al. 2012) we describe a low–level force control based on a model of the hydraulic actuation of the HyQ robot, which results in very good performance in the force and position tracking at the joints. Actuators at the joint can then be abstracted as reliable torque sources, enabling the use of inverse dynamics; the paper illustrates how the robot is then able to accomplish challenging tasks such as very fast and dynamic movements. Inverse dynamics allows to lower the position control gains without sacrificing much position accuracy, which makes the robot more compliant hence capable of withstanding the impacts of a squat jump. Note that the controller described in this work applies the model–based approach twice: it uses a model of hydraulic actuation and the rigid body dynamics model.

Inverse dynamics control for fixed base robots such as manipulators is well understood, efficient algorithms for this task exist and they are supported by our code generation framework (see Section 5.1.2.2 and 6.2.1).
On the other hand, inverse dynamics for floating base robots with constraints, like legged robots, is not solved and is a topic of recent research. The main issues are due to (a) discontinuity in the status of contact points (e.g. feet with respect to the ground, during walking), (b) idealization of such points as kinematic constraints even though slippage may occur, and in general (c) underactuation (there is no direct actuation on the robot base). Computing the joint torques that lead to the desired accelerations with constraint forces that also depend on the same torques is another issue.
In (Nakanishi et al. 2007), the authors use floating base inverse dynamics on a humanoid, to realize compliant control compatible with the use of pattern generators for locomotion. The paper refers explicitly to the algebra and the algorithms described in the Featherstone's literature, which is also the main reference for our code generation infrastructure (see Section 3.3). The authors develop a control law from the idea of estimating the contact forces using the joint torques at the previous control cycle, and then show the equivalence with the hybrid dynamics algorithms for floating bases described in (Featherstone 2008), treating contact forces as external forces. Simulation results on a biped robot showing improved position tracking are finally presented.

This approach requires knowledge of the contact forces, which are difficult to measure reliably with sensors and whose estimation leads to analytically incorrect solutions. For this reason, the authors proposed a new approach based on the QR decomposition of the constraints Jacobian, which yields dynamics equations that do not depend on the constraint forces, yet describe the full dynamics of the system (Mistry, Buchli, et al. 2010). This technique is developed on the assumption of a properly constrained robot, which is the condition allowing the projection of the dynamics equations into a space of coordinates of reduced size. See also (Mistry, Nakanishi, et al. 2008), which discusses the role of constraints in resolving the under actuation issue of floating base robots, to devise a controller based on inverse kinematics.

A perhaps obvious remark we find, however, worth mentioning, is that from the point of view of the domain analysis and the software design, focusing on the tasks of the robot rather than on its joints is a natural approach stemming from the abstraction principle (Ghezzi et al. 2002). In other words, a software engineer with no specific background in the topic, when asked to develop some programs to control a robot, would likely start modelling components for possible tasks – like pointing the end–effector somewhere – with the aim of hiding in lower–level components the actual actions to achieve such tasks, i.e. controlling the joints.
However, the literature shows how this abstraction in robotics is far from being straightforward, because joint space control alone already exhibits several difficulties; the operational space formulation however, might provide a way to realize in software such a separation of concerns.

3.3 Rigid body dynamics

There exist different approaches to solve the dynamics of multibody systems such as articulated robots, but research in the field has developed and identified the most effective and efficient *numerical* procedures for the purpose, which is what we are interested in for our work. Our code generation framework described in Chapter 5 and 6 is entirely based on the algorithms and the *spatial vector algebra* described in the works by Roy Featherstone. He has carefully investigated the field, by analyzing the asymptotic complexity of existing algorithms, inventing new ones, describing guidelines for efficient implementations, and providing the community with a comprehensive review about spatial vectors (Featherstone 2008, 2010a,b, 2013).
His book about rigid body dynamics algorithms includes much of such work, the main matter being the algorithms for solving the forward and the inverse dynamics problems (Featherstone 2008). First, it introduces the formalism of spatial vectors, and the main equations of the physics of rigid bodies expressed in such formalism. Spatial vectors are six dimensional coordinate vectors that can represent the generalized velocity (*twist*) and the generalized force (*wrench*) for a rigid body. The main reasons behind the use of spatial vectors lie in the compactness of the resulting notation, which has fewer equations and quantities if compared to 3D vectors algebra, but also in their expressiveness, so that reasoning about the physics of rigid bodies is easier if one is used to spatial vectors (Featherstone 2010a,b).

The book also includes a detailed chapter about the modelling of kinematic trees, which is also the basic material for the derivation of the software model presented in Section 5.2.1. The author then provides an exhaustive description of some of the most efficient algorithms for dynamics, with pseudo–code implementations adopting the spatial vectors notation; see Section 5.1.2.2 for a brief review of them, and also Section 6.2.1.
Other topics of the book include closed loop and floating base systems, whose support in our framework is the subject of current research.

3.4 Robotics software

According to the presentation of the joint research project BRICS (*Best practice in RobotICS*), which aims at identifying best practices in the development of robotics systems, such development process often lacks a rigorous structure and principles (Bischoff, Guhl, et al. 2010), even after decades of research in the field. A typical example is software development for robotics, where the lack of design and identification of effective abstractions lead to the development of code–driven systems as opposed to model–based ones. As a matter of fact, software engineering *for* robotics has only recently become an explicit research area (especially if considering the age of the two disciplines), as shown for example by the birth of a new journal (Brugali 2010).
In this context, in (Schlegel et al. 2009) the authors point out the gap between the experience available in robotics and the exploitation of such knowledge for a proper software development process. They describe the need for model–driven development to tackle the complexity of robotics software and relieve roboticists from hand crafted development, which is expensive yet not very effective. This point is also an important motivation of the work described in this thesis.
In (Steck et al. 2010), starting from the same premises, the authors focus on the importance of resource awareness and non–functional requirements in robotics applications, to enable, for instance, automatic run–time selection of the components to activate as a function of the available computing resources.[2] A development process, a meta–model and a toolchain based on Eclipse focused on such features for robotics systems are then presented.

The techniques of meta–modeling and domain specific languages are exploited in (Reckhaus et al. 2010) to design a programming environment independent of the target robot, to facilitate the specification and reuse of control programs. In (Klotzbücher et al. 2010), the authors present an execution environment based on the scripting language Lua, to support the implementation of internal DSLs for modeling expressive state machines for robot coordination. The work focuses particularly on dynamic memory management, in order to respect the real–time constraints during the interpretation (execution) of the state machines.

Another example of the use of a DSL in robotics, as a consequence of the need to find higher abstractions to drive software development, is presented in (Bordignon et al. 2010), which targets the specific field of modular robots. Here the

[2] We also believe this feature is essential for sophisticated control of robots, and some remarks about this point, with reference to the HyQ robot, will be given in the future work section of this thesis (see Chapter 4 and Section 8.1).

authors give an extensive description of a domain specific language for modeling the kinematics of individual robot modules and their possible interconnections, which is exploited to generate code for both the Webots simulator and a custom platform for the execution of real experiments. In the same context, (Schultz et al. 2007) presents a high level language built around the concepts of roles to facilitate the programming of controllers for the modular robot ATRON, independently of its physical configuration. While sharing the approach of model based generation and the focus on kinematics, our work targets the different domain of robots with linear or branched structure composed by rigid links (such as manipulators or legged machines); it focuses on the generation of efficient dynamics algorithms applicable in different components of a software framework for robots.

In (Nayar et al. 2007), the authors describe a software framework designed to generalize modeling and control of a variety of robotics platforms, such as wheeled rovers possibly with manipulator arms. One of the aims of the framework is to devise a general model (i.e. a data structure) for the mechanisms to allow interoperability with diverse algorithms, such as forward and inverse kinematics. The modeling of kinematics trees is almost the same as the one described in this work (see Section 5.2.1). The implementation is quite different though, as they use a regular hierarchy of classes so that robot models have to be specified programmatically and the related algorithms have to explore such data structures at run–time, while our approach is based on code generation that exploits the mechanism model offline (i.e. before the actual start up of the robot software). Our software shares with (Nayar et al. 2007) the aim to be general and adaptable to a wide class of mechanisms, but it is explicitly designed to be independent of any framework or middleware in order to be reusable in diverse contexts.
In (Brooks et al. 2005) instead, the authors address the more specific field of mobile robots – meaning robots with wheels or tracks typically equipped with a variety of sensors like cameras and laser range finders – and specifically the benefits of Component–Based Software Engineering for this kind of applications. Compared to our work, this paper addresses a different class of robots (even though also articulated robots, the subject of our work, can be mobile) and a different level of software integration (deployable components versus specific algorithms for kinematics and dynamics); however both works share principles like the need of addressing the complexity of robotics software, and the importance of the reuse of existing implementations.

An interesting work in the field of software for robotics, specifically about the problem of robustly implementing fundamental concepts in robotics such as positions, velocities, coordinates, can be found in (Laet, Bellens, Bruyninckx, et al. 2012; Laet, Bellens, Smits, et al. 2012). These articles address the issue of the ambiguities of existing implementations regarding the geometrical relations related to rigid bodies, which are ubiquitous in robotics software: position and orientation (pose), velocities, forces, and their coordinate representation.
Such ambiguities arise from the lack of standardization of notation but especially in the implicit assumptions libraries are based on, assumptions that determine incompatibilities, high costs for system integration and the risk of implementing non–physically consistent operations. The first article aims at identifying

the entities required to uniquely and unambiguously specify the *semantics* of geometric relations and at defining a corresponding terminology; the second one describes a proof–of–concept implementation that can be applied to existing geometric libraries and can provide consistent semantic checks of the geometrical operations, signaling for example an incorrect composition of two twists.
These articles are explicitly focused on a rigorous and exhaustive formalization of geometric relations and on a dedicated implementation, while the present work targets a wider spectrum of concepts with the main purpose of easing the development of simulations and controllers. However, they share the fundamental rationale in which robust software has to explicitly expose all the properties required to uniquely identify the objects it manipulates, e.g. coordinate representations of velocities.

In the robotics control literature, model–based control has been extensively discussed, and some examples have been given in a previous section. However, almost all the works address the theoretical aspect of the controller, rather than explicitly focusing on the software required to implement the approaches. Maybe the most notable exception to this is the Stanford Whole-Body Control (S-WBC) open source project (Philippsen et al. 2011), initially released in 2009 as a result of the efforts of the Stanford Robotics Lab to bring the operational space formulation into a software for the community.
The S-WBC comprises a library for joint-space kinematics and dynamics, on top of which additional components provide the abstractions required by operational space control. Specific effort has been spent on the configurability of the framework, so for instance classes can expose their own parameters via a reflection mechanism, and these can be changed at runtime.
Even though our approach shares with the S-WBC some design principles and goals such as flexibility and robustness, we aim at software components that are even more general and reusable, since they have a narrower domain, and not at a full–featured framework. We are also particularly concerned with efficiency and speed of execution, for scenarios with hard real–time constraints. The precise set of features and the lack of dependencies on middlewares or specific technological platforms make our software suitable as a building block for other applications, of which the S-WBC itself might be an example.

3.5 Robot modelling and code generation

The generation of code exploiting symbolic math engines is not a new idea, and some examples of it can be found already in older literature. For example, the premises of (Toogood 1989) are analogous to some of the motivations of our work: basically, the need for efficiency and the intrinsic difficulty of implementing the dynamics of a manipulator by hand. The paper describes two algorithms for inverse and forward dynamics and a computer program capable of symbolic manipulation and simplification that generates a Fortran implementation of such algorithms.
The improvements of our work with respect to this previous approach are multiple, and include: we base our code generation on state-of-the-art algorithms which are recognized to be the most efficient for the numerical solution of dynamics (see Section 3.3). Then we base our software on generic software models

that capture the main features of the domain of interest (e.g. the kinematic tree abstraction), which make the system more flexible, reusable (but also more understandable for other users). Finally we exploit recent, effective technologies like the Domain Specific Languages (Fowler 2010) and Java, which also make our code generation more easily usable by different people.

SD/FAST (Sherman et al. 2013) is a powerful software that produces C or Fortran implementation of the equations of motions of any given mechanical system which can be described as a set of rigid bodies connected through joints, possibly with further motion constraints. SD/FAST is quite flexible, in that it supports a wide selection of topologies, of joints, simulation conditions (forces, prescribed motions, etc.). Similar to our work, this software is not a fully fledged simulation language or environment, but rather aims at being a useful tool for the development of a simulation process, by providing the equations of motion that would be particularly difficult to write by hand. To this extent, our work is designed to be even more reusable in different contexts, since it can target multiple programming languages, it is based on neat models that make very explicit what the generator is doing, it is not tied to specific frameworks or technologies (besides Java, Xtext and Maxima, which however are open source and run on a variety of platforms – see the introduction of Chapter 6). Moreover, it is also capable of generating consistent, hard real–time capable code which can be actually deployed on a real robot controller, therefore going beyond the realm of simulations. SD/FAST, on the other hand, appears quite tied to simulation applications only; in addition, it does not seem to be maintained anymore and the available release is a bit old.

Similarly, Robotran (CEREM 2013) deals with the dynamics of multibody systems; after reading a user model defined with a graphical editor, it can output symbolic equations of motion and perform simulations interacting with MATLAB. Robotran is quite sophisticated and can handle a variety of mechanisms, including closed loop assemblies and vehicles. It includes its own symbolic engine that allows the generation of optimized code in C language. Simulations though, have to be performed with MATLAB and Simulink, which refer to the C code, and perform the numerical integration. So Robotran is tied to these programs, and does not support explicitly the generation of standalone C code – although one can try to extract some routines out of it.
Even though Robotran can target more complex mechanisms than open kinematic trees, our approach still provides some advantages. It is based only on open source technologies and has limited requirements to be used; the explicit models underlying its implementation make it easier to understand its behavior and therefore re–use its output in whatever application, which also includes hard real–time robot controllers, with code that exhibits no external dependency but the Eigen library (see Section 6.2); Eigen and the language features even make the generated code much more readable and neat with respect to low level C code; it may seem a minor point but it fosters reuse. More programming languages can be addressed as well. In general, our framework can be extended to support closed loop systems by further developing the domain model (see Section 5.2.1.3 and 8.2.2) and adding the other algorithms described in (Featherstone 2008).

SL is a rigid body dynamics simulator and robot controller package – we are currently using it for our research (see also Section 4.2.4) –, which uses a custom format for the description of the kinematics and can generate highly optimized

C code for kinematics and dynamics (Schaal 2009). The performance of such code is definitely high: the SL design started roughly twenty years ago with the purpose of doing real–time control on articulated robots, with the computers of that time.
Although SL already meets some of the points that motivate our work, and as such it provides a truly valuable support for the development of simulations and controllers, much improvement can be introduced with regard to the code generation process. Among other things, SL depends on a commercial software, Mathematica, that is used to parse the text file with the description of the robot and to perform the symbolic calculus required to generate optimized code; this file is in fact much harder to read with respect to the language we propose in this work, and also the syntax errors reporting is not very effective.[3] In addition, the generated code is tightly coupled with the rest of the SL infrastructure, and it would be extremely hard to reuse it in a different application; it is not customizable (i.e. to generate code for some different coordinate transforms) and it is only in C language.
Other information about SL are given in Section 4.2.4.

Some examples concerning the modeling and the description language for multibody systems can also be found in existing packages and programs. In the Robot Operating System ROS (Quigley et al. 2009), for instance, robot description files need to be provided with a custom format based on XML (the URDF file format) that makes it harder to read and maintain as compared with a dedicated solution such as a DSL; in the OpenHRP simulator (OpenHRP group 2013), the language for the models comes from the 3D modeling field, and mixes graphical aspects and sensors with kinematics parameters. In addition, to the best of our knowledge these languages do not come with efficient code generation capabilities.
Another example is Modelica, a multi–domain, object–oriented modelling language for a variety of physical systems used also in industry (Modelica Association 2013). Its models basically contain the system equations, which then need to be transformed into executable code or into a form suitable for a simulation engine. The main advantage of Domain Specific Languages such as the ones we propose in this work, is that they are concise and easy, since they are dedicated to one purpose. On the other hand, Modelica is a very general, multi–domain language that may prove inconvenient for very specific requirements (the version 3.3 of the language specification is 282 page long), such as code generation tailored for robotics.

The idea of using code generation to achieve more efficient software is not restricted to the robotics domain. For instance, in (Mattingley et al. 2012) the authors propose to generate an optimized implementation of solvers for convex optimization problems, so that the solver can execute faster and within some hard real–time deadlines. The authors call *family* of the problems the set of all the same optimization problems that depend on certain parameters. A problem *instance*, on the other hand, is identified by a specific value for all such parameters. The idea is to exploit ahead of time the information about the family (i.e.

[3] The file format for the description of robot models in SL happens to be an *internal* DSL based on Mathematica scripts. See Section 2.3.

common features of all the possible instances), in order to generate specialized code which works efficiently only for such problems. The underlying idea is very similar to what is described in this work; a slight difference purely related to nomenclature is worth mentioning. In this work we deal with optimized solutions for specific robot kinematics, which we think of as an *instance* of the kinematics model, while in (Mattingley et al. 2012) they address the family of problems, since a solution for a problem instance would not make sense.

Chapter 4

A computing system for the control of articulated robots

The purpose of this chapter is to explain in detail the computing system currently running on the HyQ robot (see Section 3.1 for a description of it); the explanation aims at giving the reader a clear idea of what is required, in terms of software and hardware, to bring a complex robot into the state in which preliminary but also more sophisticated experiments can be performed.
HyQ is a quadruped robot that is about one meter long and 70 kg heavy, with electric and hydraulic actuation. The powerful actuation system and the challenging goals of the project make HyQ a very good example of a robot that needs a robust, reliable and efficient software which is also inevitably going to be quite complex. For example it is critical for certain computational modules to be very reliable (see also the paragraph about real-time 4.2.1) since certain failures while using the hydraulic actuation are potentially dangerous for the human operators and the robot itself.

For these reasons the HyQ robot is taken as a representative example of the class of sophisticated, articulated robots, but the information included in this chapter is of general value.
The chapter starts with an overview of the current computing system, and then describes general issues such as real–time computation and each single block of the system with greater detail.

4.1 Overview

This section gives a general overview of the computing system currently available on the HyQ robot, covering both the hardware and the software aspects. More details about each part will be given in the following section.

The diagram in Figure 4.1 shows the main software blocks implementing the control of the robot, and gives a rough idea of the role of each one with respect to the whole. The purpose of the diagram is to show both the organization of the software on HyQ but also a quite general layout that can be applied to a variety of robots with analogous features.
This organization by no means represents the definitive solution to build a complex system that controls a real robot and makes it autonomous, as it has

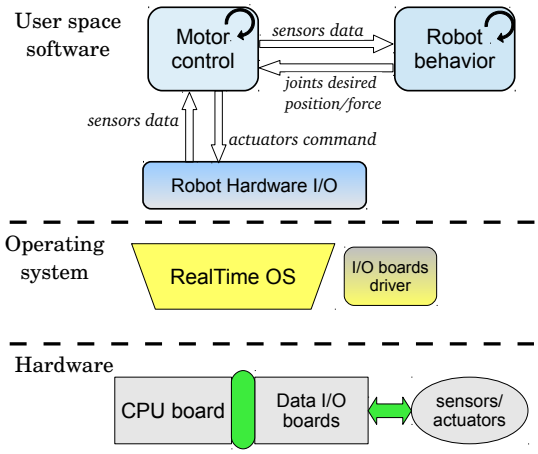


Figure 4.1 – Logical view of the current computing setup of the HyQ robot. Data acquisition boards enable the interaction with the robot hardware (e.g. sensors); the green parts represent physical connections. The real–time capable operating system includes a driver giving access to the facilities of the I/O board. The user level code includes a library to abstract the hardware and two other active components: the motor control module and the robot behavior module, both running as processes. The sensor data include position and force measurements at the joints, while the trajectories contain desired positions, velocities and forces for the joints.

some limitations that will be discussed in Section 8.1. However it is already effective for trying complex behaviors with the robot and address open research questions (such as quadrupedal locomotion), i.e. it is a representative system that serves to illustrate the challenges and our solutions. We implemented such an architecture on our robot and achieved promising results (see Section 7.1).

At the base of the diagram of Figure 4.1 we find the hardware and the operating system, which obviously host all the rest of the software modules. A general purpose computer board is equipped with input–output (I/O) boards that provide the means to communicate with the other electronic components of the robot (e.g. sensors – see Figure 4.2). Even just this simple description reveals the important deployment choice of using a single computer running the whole system, a point that will be further discussed at the end of the thesis, in Section 8.1.
The most important requirement for the operating system is to be *hard real–time capable*, that is, capable of dealing with processes which need to be triggered with precise and reliable timing, and which must not be interrupted by other processes in the middle of their execution.

The next module depicted in the diagram is the hardware abstraction layer (HAL), or the hardware I/O layer. In this case the term hardware refers to the robot hardware, that is all the electric and electronic devices that are necessary to enable the automatic control of the robot. These devices are typically *sensors* that provide numerical information about the status of the robot and the environment, and *actuators* that provide the mechanical power required to move the links of the robot. The terms input and output refer respectively to

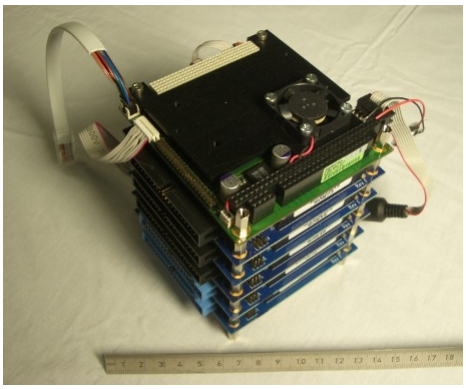


Figure 4.2 – The computer with the data acquisition boards currently running the control software of HyQ. The black board on the top is a regular Pentium–based motherboard compliant with the PC104 standard, while the other blue boards (Sensoray 526) stacked below provide input and output capabilities for both digital and analogue signals.

the sensor measurements being transmitted to the computer, and the actuators command flowing from the computers to the drivers of the actuators.
This layer is entirely implemented in software, but it obviously must communicate with the I/O boards that provide the actual, physical connections to the devices and in fact constitute the boundary between hardware and software. The main purpose of the HAL is to abstract from the low level details of the communication with the hardware and provide a neat and clean interface to the numerical algorithms which do the actual control.

The following two modules implement the lowest level numerical algorithms which allow to move the robot according to user–defined trajectories. The first one is the primary consumer of the sensory data, and it is mainly responsible for the so called *motor control*. This term basically refers to the processing of the input data coming from sensors and other external modules, in order to produce the appropriate output commands for the actuators, in a closed-loop fashion. Probably the most common and simple realization of this module is a PID (Proportional, Integrative, Derivative) controller on the position of the joints of the robot.
The second module – *robot behavior*, in the figure – generates in turn commands for the motor control module, that is, it provides directives about how to move the robot. One might think of this module as a trajectory generator, that is some sort of machinery which determines a set of desired positions (i.e. joint angles) at each time step (adopting a certain degree of discretization of the physical time). However this term is a bit generic and may hide quite a broad set of submodules interacting according to complex patterns. It is indeed quite intuitive that providing directives about how to move the robot is something that can be done in a variety of different ways, from very simple algorithms (e.g. arbitrarily impose a sinusoidal trajectory to each joint of the robot) to approaches involving complex reasoning (e.g. using input data from stereo cameras, or exploiting probabilistic models).
Unless otherwise specified, we will refer to this module as the *task* module, to

mean the part of the system which has to determine which movements the robot should perform. Also because this module can be realized in many different ways, it is highly desirable for the motor control to be as much as possible agnostic with respect to it, and limit the interaction between the two modules according to a simple and specific interface.

The next section will give some more details about the components mentioned in the previous paragraphs.

4.2 Key aspects of the platform

4.2.1 Real–time

The term *real–time* is typically used to describe computations whose correct outcome, which in principle can be anything, does not depend only on the correctness of the implementation of the algorithm, but also on the actual timing of the execution of the same implementation. The timing usually refers to a deadline, a duration, which should not be exceeded for the result to be usable and also to guarantee the proper functioning of the overall system.
It is also very important to realize that, in principle, real–time does not imply the requirement of being as fast as possible, since the actual constraints of a compliant implementation depend solely on the characteristics of the specific application domain, with reference to time constants. The fundamental point of real–time computations concerns the *guarantee* that a certain function can be executed within a certain time interval, whatever the length of such interval.

A typical example is software for the decoding of compressed audio and video streams, i.e. media players. Most of the time videos contain data which cannot be directly visualized because it is compressed in some form in order to save disk space. At the same time it cannot be uncompressed entirely at the beginning because of memory limitations. Therefore all the computations required to decompress the streams and then visualize the video (and play the audio) must happen *on line*, that is while the video is being displayed. It follows that such computations must happen at a speed that is compatible with the *physical time* constraints imposed by this specific application, for instance display on the screen 25 frames per second. In other words, all the operations should take less than 1/25 seconds to be executed. The case of multimedia is a well know example of *soft* real–time, which means that the consequences of not meeting the deadlines can be acceptable up to a certain extent (which again depends heavily on the domain). For example, it does not matter much if some frames are not exactly displayed for 1/25s each (because the human eye/brain would not perceive it), and if there are occasional delays during the visualization.

On the other hand, *hard* real–time refers to application domains in which respecting the time deadlines is a critical requirement, since otherwise the result of the computation might be completely useless or even lead to severe consequences. This is for example the case for robot control, which is what we are going to discuss here and in general the application domain we will be referencing throughout the whole text.
A perhaps obvious but fundamental difference between robotics software and standard software (e.g. a media player or a warehouse management applica-

tion) is that we do not simply want it to provide us with some meaningful results (whatever they might be), but we want it to make *a robot* perform some meaningful action. In a first approximation, the gap existing between purely numerical results – typically delivered in some form to a human user – and concrete actions performed by a mechanical system, is filled by implementing control theory techniques (see Section 2.2).

For a variety of factors – whose description would go beyond the scope of this work – the proper functioning of control techniques fundamentally depends on the frequency of their execution, as well as the time accuracy of their orchestration. Hence their implementation and the execution environment must be real–time capable.
Intuitively, a controller should be real–time because its purpose is to influence a concrete system that lives in the physical world and evolves according to its own dynamics (e.g. a moving rigid body); such dynamics must be observed and influenced in a timely manner for the control to work properly. For example, if a joint controller does not sample regularly the status of the joint, the link of the robot might keep moving beyond the desired position before any corrective action is taken. The frequency of the controller (i.e. its speed) depends on the application domain: how fast the motion of the link needs to be changed, which is the frequency of possible disturbances to be rejected.

Therefore a critical part of the software for real robots, as in the case of HyQ, is the implementation of control algorithms, as for instance a PID controller for the positions of the joints of the robot.
Real–time capability is a critical issue for the control of real robots, since a failure in the controller (which also happens when a deadline is not respected) would lead to unpredictable effects on the robot like random movements of the links (at random speed) which might be harmful for the robot itself but also for human users working in the surroundings. This is really a critical point of the system development especially when working with powerful robots such as HyQ, which in order to fulfil the long term goals of the research project has been designed with an actuation system that can deliver a lot of power, enough to seriously hurt a person.

4.2.1.1 The Xenomai framework

In order to satisfy the requirements described above, it is then mandatory to have computers which are capable of running processes with real–time guarantees. In principle, a very simple micro-controller executing periodically a single numerical algorithm could very well be considered as a real–time machine, assuming its clock to be as fast as needed for the specific problem. This is because the duration of the computation is fixed, there is no concurrency between multiple processes and so no rescheduling, and therefore the frequency of execution of the routine would be guaranteed.

However, when developing entirely a new robot it is desirable to have as much as possible standard platforms and development tools, to reduce to the minimum the overhead due to the issues inherent to more elaborated solutions with custom components. For example, using an off-the-shelf computer with a general purpose operating system, allows to maintain all the code on a single machine, equipped with standard facilities and tools that drastically ease development, building and testing (including systems for version control, which

would be much harder on diverse hardware platforms).

A Linux operating system, for example, can be enriched with Xenomai, which is a patch capable of bringing real–time capabilities to the existing kernel (Gerum 2004; Yaghmour et al. 2008) thus allowing to use it for the control of robots. Xenomai is a so called co–kernel that lives next to the regular Linux kernel and it is responsible for the management of the real–time tasks running on the machine. This architecture is enabled by an additional software component called I–pipe, which sits between the hardware and the kernels; it basically overrides the hardware interrupt mask provided by the CPU with a set of virtual masks, one for each running kernel (called "domain"). These kernels (typically Xenomai and Linux) are assigned a fixed priority (Xenomai having the highest), which determines the order interrupts and other hardware events are notified by the I–pipe to the domains. The highest priority domain (Xenomai) can

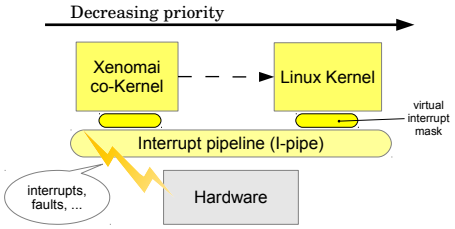


Figure 4.3 – A simplified architecture of a Xenomai–patched Linux system, which has its foundation in the I–pipe component for the prioritized delivery of events, such as interrupts. The dashed arrow refers to the dependency of Xenomai on the Linux kernel for non real–time services like fault handling and regular system calls.

handle events very quickly first, possibly passing some of them to lower priority domains. This also allows Xenomai to rely on all the existing Linux facilities (such as handlers of exceptions) without need to reimplement them. See Figure 4.3 for a simplified view of the Xenomai architecture.

From the user's perspective, a very convenient feature of Xenomai is the capability of running real–time tasks in *user space*. This term coming from the operating systems theory refers to the execution mode of regular applications which features a limited set of privileges and a separate memory area. *Kernel space*, on the other hand, is the execution mode for the kernel of the operating system where everything is allowed. A mistake in some kernel space code (such as a kernel module providing some more functionality to the operating system) may lead to an unrecoverable system failure, making development of kernel space applications very inconvenient. If real–time capabilities are available in user space, then developing and trying new real–time applications such as a robot controller becomes much less demanding.

4.2.2 Communication with the robot hardware

The real–time capabilities described in Section 4.2.1 are required to make the software reason meaningfully about the physical environment as far as the time is concerned. For example to react to a slippage of a leg of the robot within a

certain time threshold, or to make sure a control loop is running at the proper frequency.
Another fundamental issue in the software of a robot is how concretely data is exchanged with the available hardware components like sensors (just hardware, hereinafter). There must be a way for the software to access the information about the status of the robot and the surroundings (input), as well as a way to turn some of the numerical results into physical signals that affect the actions of the machine (output). Very common examples of these points are the reading of the current position of the joints, to make the control logic aware of the positions of the robot links, and the sending of voltage signals to the electric motors that move the links.

To enable this interaction between the two different domains (hardware and software) it is necessary to have components lying at the boundary and acting as a bridge, transforming numerical values into physical signals and vice–versa. This is the realm of micro–controllers, data acquisition boards, DSP (Digital Signal Processing units), FPGA (Field Programmable Gate Arrays), etc. An exhaustive description of these alternatives and of their distinctive features goes beyond the scope of this work, so we will give only a few examples with particular reference to the solution adopted on the HyQ robot.
In general, these components are required to have the electronics for the input/output of digital and analogue signals, like ADC (Analogue to Digital Converters), digital counters to read encoder signals, simple electric connectors to read and send digital bits, and so on. The actual set of features obviously depends on the requirements of the project. In addition, the component should expose some sort of digital interface to configure and to access its functionalities by means of a computer program. The actual hardware/software boundary often lies in a chip equipped with a micro architecture, i.e. capable of interpreting numerical codes received on a bus as instructions. See Figure 4.4.

The control system of HyQ is equipped with five Sensoray526 general–purpose I/O electronics boards which provide exactly the aforementioned features. They are compliant with the PC104 standard so they can be physically stacked on top of each other together with the CPU board. The communication with software running on the computer is very convenient, since the registers of the boards are directly mapped into some memory segments, so that reading and/or writing to a particular location in memory of the computer results in the actual read or write of a register of the boards. Registers are used for configuration, to trigger certain functions such as an analogue to digital conversion, and to read back results. In this way the Sensoray526 is implementing the actual hardware–software bridge required for the input–output subsystem.

Once a hardware technology for the I/O is chosen, typically one needs to develop some sort of *driver* that allows client code to access the facilities of such hardware. In general, the purpose of this pure software module is to abstract the details of the underlying device, relieving the client code and therefore the developer from knowing its internals.
The device driver obviously fundamentally depends on the hardware it is targeting and on the environment it is going to run in (e.g. the operating system of the computer), and it is a low–level component of the whole architecture[1].

[1]With "low–level" we refer to the conceptual localization of a component of the system (e.g. a software module) addressing issues that, in comparison to the general domain of the

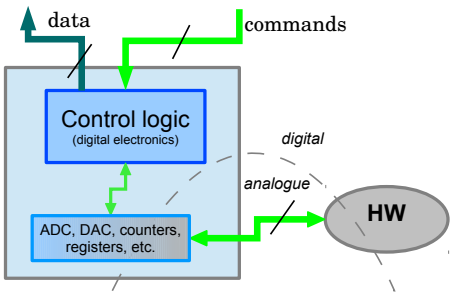


Figure 4.4 – An abstract schema of the components enabling the communication between the software and the hardware of a robot, like sensors (e.g. analog temperature sensors, digital encoders). The arrows represent *physical* connections, i.e. electric wires; the blue boxes are composed of digital electronics and components at the boundary of the two realms, i.e. analog/digital converters; the *control logic* block is essentially a CPU that interprets some signals as commands and triggers the other components. Such commands are digital as well as the data flowing in the opposite direction: they are produced and consumed by other components with a similar structure, e.g. a computer where the control logic is a powerful CPU running higher level software.

On HyQ this functionality has been implemented with two software components specific for the Sensoray526 board (see Figure 4.5):

- the actual device driver, which is a module for the Linux operating system with Xenomai, compliant with the Real Time Driver Model (Kiszka 1997). It is a compact module written in C that basically hides the low–level memory reading and writing to access the actual registers of the Sensoray526 board.

- an additional software wrapper further abstracting the low–level interface available in the driver, providing in turn another interface that is much more intuitive and closer to the actual functionalities available on the board. This code was developed in C++.

A fundamental requirement of these software pieces, as well as of any other additional components that may be used within the processes implementing the robot control, is to be compatible with the constraints of real–time execution. This basically means that the entire code these modules are comprised of must not rely on any operation whose execution time cannot be guaranteed, such as general purpose I/O operations.
In other words, if on one hand the deployment environment (e.g. a certain computer with a certain operating system) must be capable of providing real–time guarantees, on the other hand the user's software itself must obey strict constraints to ensure that the time deadlines *can* actually be respected.

application (e.g. robot grasping), are less abstract and/or more related to technological details.

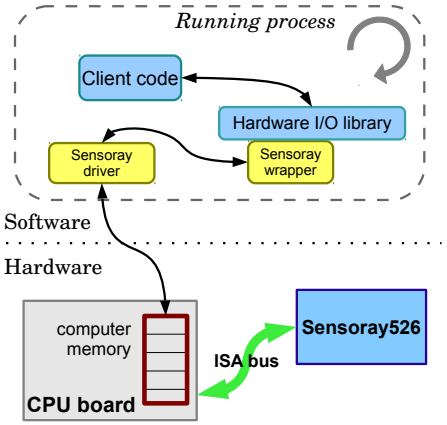


Figure 4.5 – The software components used to provide a convenient access to the facilities of the Sensoray526. The ISA bus maps the registers of the device into the computer memory, hence low level I/O operations performed by the driver boil down to reads and writes into the memory. The *wrapper* introduces additional abstraction exposing a higher level interface, e.g. functions to read an ADC or to raise a digital output port. These functions are used in turn by a library, described in Section 4.2.2.1. Thanks to these components, I/O applications using sensors and other devices can be quickly developed on a computer equipped with one ore more Sensoray526 boards (which is indeed what has been done for the HyQ robot).

4.2.2.1 Hardware abstraction

Before focusing on the software modules performing some meaningful computations about the control of the robot (e.g. position control of the joints, attitude estimation for a walking robot, etc.), it is desirable to have yet another abstraction layer which provides a convenient access to sensors and actuators data, in a transparent way.
Basically the purpose of this layer is to prevent the dependencies on the underlying hardware from reaching higher–level components, by providing a generic interface that solely depends on the *information* that has to be exchanged with the hardware. For example, it is almost mandatory to be able to get the current angles of the revolute joints of the robots (e.g. values expressed in radians) without any reference to the actual sensors performing the measurements nor the technology used to communicate with the same sensors. This separation of concerns, or modularity, improves the maintainability and enforces the re-usability of the software (Dijkstra 1982). For example, it allows to subsequently change anything in the lower layers (from the actual sensor model to the technology for the I/O) without affecting the components above, which in turn makes it possible to exploit technological opportunities – like a new, more precise sensor on the market.

The idea behind the design of this layer is to abstract as much as possible the hardware components available on the robot (just hardware, hereinafter), focusing strictly on the *semantics* of its data. On the other hand this layer should not incorporate any application specific logic that might compromise the

reuse of the code in different contexts. It should be as thin as possible and perform only very general processing such as turning a raw value into a more meaningful sample of some physical quantity, that is the processing which is likely to be required by any possible client application.
The concrete implementation of the public interfaces still depends on the underlying layers because it involves some communication with the hardware, however none of the details should arise at the client's level. Ideally, this layer should be the last part of the system requiring an update as a consequence of modifications in the hardware of the robot.
As mentioned before, the implementation should also be real–time compatible, not to break the guarantees required for a proper functioning of the control. Even if real–time is not in principle related to absolute speed, it is also important for the implementation to be thin and efficient, since it will be executed within processes running at high frequencies (e.g. a control loop at 500Hz).

A concrete example In the following, a concrete example taken from the software running on the HyQ robot will be shown. All the code of this layer is written in C++.
The example pertains position sensors (e.g. encoders), since they are amongst the most common type of sensors used in robotics, due to the fundamental need of providing the robot with a perception of the position of its own links.
Figure 4.6 gives an overview of the classes involved in the representation of position sensors. The organization of the classes follow a general pattern that exploits the inheritance of object orientation (Meyer 1997; Snyder 1986); it has been adopted also for other sensors, encoders are just a notable example. In

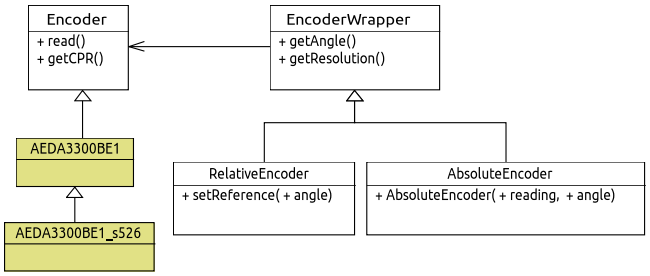


Figure 4.6 – Layout of the classes modelling position sensors, as implemented in the software platform for HyQ. The `AEDA3300BE1` is the specific model of the relative encoder mounted on the robot. An `Encoder` gives just a number, while an `EncoderWrapper` gives an actual angle. Note that the relative encoder is configured with a function call (that can happen at any time), while the configuration of the absolute encoder happens only once in the constructor (i.e. at the startup of the application).

particular, the left part of the diagram shows the separation between the higher–level class, an abstract encoder with a minimal interface, and the other classes that instead exhibit dependencies on the actual hardware of the robot. These classes (colored in the diagram) are further partitioned to represent the specific model of sensor and the technology used to communicate with it (the Sensoray526 in our case).

The former (`AEDA3300BE1` in the figure) wraps information and details related to the sensor itself, which often are simply factory parameters; a typical example for an encoder is its resolution. The latter instead (`AEDA3300BE1_s526`), contains the actual code to read data from the sensor and hides the details of the communication; in our example, this class interacts with an instance of the wrapper class of the Sensoray526 (see above). This is the only class that would be replaced should the underlying I/O mechanism change.

The purpose of the classes `Encoder` and `EncoderWrapper` (on the top of the diagram) is to separate the modelling of encoders purely as displacement sensors and as actual angle sensors. The former is therefore simply the software representation of the hardware device, abstracted as a provider of a single measurement. The latter instead adds a bit more information to the measurement by converting it to a meaningful angle, expressed in radians, according to its configuration parameters (which are determined elsewhere). The encoder wrapper is independent of the actual type of the encoder instance providing the raw measurement.
The separation of concerns principle stressed in this design makes the implementation modular and provides significant advantages. The classes are lightweight and self contained, and it is easy to change a component without affecting the others. This does not mean that, for instance, changing the technology for the communication with the hardware would result in trivial modifications of the software, but at least the affected components would be limited and the need for fixes would not propagate much.

Table 4.1 shows the actual interface for relative encoders and absolute encoders as implemented for the HyQ robot. The former measure the displacement – in the form of a rotation or a translation – with respect to the initial position at the start–up of the system (i.e. when the sensors are powered); the latter instead give a value which depends solely on the absolute physical configuration of the sensor (e.g. the position of an axle with respect to the casing of the sensor), and therefore does not change from time to time. The interface is minimal,

Relative encoder	Absolute encoder

`double getAngle();`
`double getResolution();`
`void setReferenceAngle(angle);`
	`AbsoluteEncoder(rawValue,`
	`angle);`
	`double getAngle();`
	`double getResolution();`

Table 4.1 – The interface for the class representing relative and absolute encoders.

very simple, and basically includes only a function to read an angle. The class itself is independent on conventions, ranges etc. about the angles, which is an information that should be encoded elsewhere, but instead it takes parameters to be able to turn the relative readings from the sensor into angles meaningful for the application.
In particular, the relative encoder takes an angle of reference (which must be already expressed in the proper units, e.g. SI units) so that relative measures taken afterwards can be turned into the absolute angles that are much more

useful for the application. The formula is very simple:

$$\theta = \theta_{ref} + (raw - raw_{ref}) \cdot R$$

where θ_{ref} is the angle set by the client code, raw is the current raw reading of the sensor, raw_{ref} is the reading at the instant in which θ_{ref} was set, and R is the resolution of the encoder (expressed in radians, assuming that the raw readings of the encoder are dimensionless). θ_{ref} must be set by a run–time call of the corresponding setter function, since the actual value can be determined only at run–time and the user might want to use this function more than once, to recalibrate the sensor.
On the other hand the absolute encoder provides a *constructor* with the required parameters, to emphasize that these values do not change during the life of the application (i.e. they are *configuration* parameters). As a matter of fact such parameters depend purely on the physical layout of the sensor when mounted on the robot, and on the conventions about the joint angles as desired by the user. The configuration parameters are a raw reading value and the corresponding physical angle, and they allow to turn any other reading of the absolute encoder into an angle. The formula is exactly the same as before. The difference, as already mentioned, is that raw_{ref} and θ_{ref} do not need to be determined at run time.
A typical use of absolute encoders is to read the absolute joint angles of the robot at the start up of the system, and then use such a value to configure the relative encoders (typically much more precise and therefore more suitable for the control of the robot). This approach relieves the user from moving the joints to a known position (e.g. at the mechanical limits) every time the system is started, to manually calibrate the position sensors.

4.2.2.2 Robot specific layer

Note that the software for encoders described in the previous section, presented as an example of a component of the hardware abstraction layer HAL, is quite general and does not depend on any specific robot. This reflects the obvious fact that some hardware (like commercial sensors) is not constrained to be used on a single machine only. It is desirable that as many components as possible share this property, since it allows the reuse of the same code for different applications, as for instance different robots with similar sensors/actuators.
An additional layer exploiting the general purpose HAL components and plugging in more specific information about the robot is therefore necessary. The content of such layer is heavily dependent on the specific robot, and its purpose is to provide a tailored I/O interface for client components which are also aware of the details of the robot (e.g. how many actuated joints there are).
This layer should have components that read and use the whole set of configuration parameters which depend on how the robot is assembled, how its electric and electronic components are wired and connected, etc.

A possible objection to this layered architecture concerns efficiency, which is critical when the system has hard real–time requirements at high frequencies of execution. However one has to keep in mind that having a number of logical layers does not necessarily imply an equal number of complex subsystems doing computations and exchanging large amounts of data. In other words such layers can be thin and smart enough not to introduce a significant run–time overhead,

but rather providing a useful partitioning of the components.
Moreover, as far as the actual code is concerned, languages such as C++ allow to enforce such a modularity in the implementation, yet providing good efficiency and speed compromises.
For example, the components described above for the encoders are very simple, and essentially more effort was required to design the structure rather than to provide the actual implementation. Everything is suitable for real–time execution at high frequencies.

4.2.3 Motor control and robot behavior

The last two blocks of the proposed architecture shown in Figure 4.1, are responsible for the motor control and for the generation of task–specific trajectories for the joints of the robot. These are the first elements of the system which are not just providing a static service to clients but instead actively perform some computations, i.e. they are implemented as *processes*.
The different nature of the operations and the (generally) different frequencies of their execution lend to the natural choice of implementing them with two separate modules (e.g. one real–time process for each) exchanging information through some sort of communication channel.

4.2.3.1 The motor control block

The primary role of the motor control block is to implement one or more numerical algorithms taken from control systems theory, which allow to drive the actuators and move the robot links as desired. The primary input information required for this purpose is the status of the robot, in terms of position and velocity of its joints. The force at the joints might also be required, depending on the control scheme (as a rule of thumb, it is necessary to sample the actual value of the quantities that one wants to control, the position of the joints being the most common example of such quantities). Other input values may include lower–level information such as the status of the actuators (e.g. the current flowing in an electric motor, the oil pressure in the chambers of a hydraulic cylinder) since they also constitute sub–systems the robot is comprised of. The motor control is therefore the primary client of the hardware I/O modules, which must provide access to all the necessary sensors and to the actuator drivers (such as the power electronics of a brushless DC motor, or the solenoids driving hydraulic valves).
Given the input, the algorithm computes the output commands for the actuators that should move the robot as desired. This procedure of sampling the system state and computing the commands happens periodically, with a certain frequency (e.g. 500Hz) and constitutes the core functionality of the component. A reliable timing is fundamental for the algorithm to be successful in controlling the actuators, therefore it is necessary to use a hard real–time process for the implementation. For the same reasons one has to ensure that all the code possibly referenced by such process (like the code for the hardware I/O) is suitable for real–time.

A deeper discussion about various low–level control approaches and their input data would go beyond the scope of this work. However, a proper understanding of control schemes, of their features, of the consequences of choosing

one or another, is important to devise a more general and effective software architecture for the whole system.
A generic control scheme is depicted in Figure 4.7 to give the reader an idea of a plausible structure of the motor control block, together with the involved data types. The scheme shows three nested control loops, each one providing

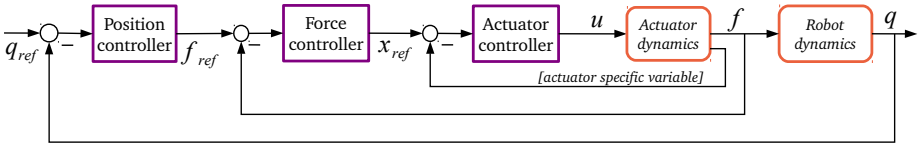


Figure 4.7 – Block diagram showing a very generic control loop. Actual control approaches may drop some blocks or introduce new ones. The suffix ref denotes a desired value a controller has to track; q represents the position of the joints, f the force acting about the respective axes. The innermost loop controls some quantity that depends on the actuation systems (it might be pressure for hydraulics, current for electric motors). The frequency of execution of the loops is determined by the controller designer and through experimentation.

the reference input for the next, more inner one. The innermost loop is specific for the actuation technology adopted at the joints of the robot, and therefore its input measurements can be very diverse (e.g. current, rotational speed, oil pressure, etc.). So is the output command, which goes directly to the actuator drivers, and could be a current, a voltage, the duty cycle of a PWM signal, and so on.
The next loops target respectively the force and the position at the joints, these also being the quantities that should be sampled by sensors. As drawn in the picture, the outermost position control loop takes a reference from an external source, and provides a force reference (i.e. the force that is desired to be acting on the joints) to the force control loop, which in turn sends its output to the actuator control loop.
All the loops as shown in the figure are purely based on *error feedback*, that is their output is a function of the difference between the reference value and the measured one. However some sort of *model based* control taking advantage of prior knowledge about some part of the system may be used (e.g. electromagnetism for electric motors, pressure/flow relationship for hydraulic cylinders, etc.). See also Section 2.2.
Note that the diagram in the figure is just an example, and different schemes are possible. For example, the current controller of the HyQ robot is based on a position control and an inner, model–based force control specific for hydraulics, while there is no further explicit control loop for other quantities (Boaventura, Semini, Buchli, Frigerio, et al. 2012).

4.2.3.2 The robot behavior block

The role of the last block included in the overview picture of Fig. 4.1 is basically generating the reference inputs for the motor control, typically in the form of desired joint trajectories, i.e. a set of subsequent desired positions (called *set*

points). Alternatively, desired forces might be produced if the outermost position control loop is deactivated.
Set points must be emitted in a timely manner possibly without abrupt discontinuities, to avoid undesired jerky movements of the robot or general instabilities in the controller. Therefore a real–time process is also necessary for the trajectory generation. However the frequency at which the set points are generated may be different from the one of the motor control, and typically a lower frequency is enough for a smooth trajectory and proper overall results. The lower–level motor control usually has to run faster, so that a few control loops are executed for each set point (again, a deeper discussion of the role of frequency in control theory is not in the scope of this work).

As discussed in the overview (Section 4.1), determining the desired joint trajectories and therefore how the robot has to move is possibly a very complex task that can be achieved in different ways. In a first approximation though, one can think of a single process implementing a periodic loop, whose core function is to compute the next set point according to some logic, chosen by the user. For example, one may use the sine function of any math library to generate the points of a sinusoidal trajectory, for some of the joints, to test the response of the controllers or simply the status of the mechanics of the robot.
This approach for the implementation of the robot behavior (or robot task), though not very extensible, is quite general and very effective to do many experiments on a robot, and for this reason it has been implemented on the HyQ robot. As soon as the logic for the specification of the behavior of the robot becomes more complex though, for instance if some sort of artificial vision is involved (e.g. stereo cameras, laser scanners), than it is obvious that a single–process, all–in–one block for the robot behavior is not appropriate anymore and a more effective architecture and implementation have to be devised. See Section 8.1.

4.2.4 The SL package

The software system described in the previous paragraphs has been successfully implemented on the HyQ robot, enabling the actual experimentation after the mechanical structure was built.
The SL *simulator and motor controller package* was chosen for our system since it basically provides an implementation of the motor control and the robot behavior blocks (Schaal 2009) (see also Section 3.5). SL was initially designed for a multi–processor architecture and implements the two blocks with two real–time processes interacting via shared memory (these two processes are called respectively the "motor servo" and the "task servo"). It is entirely written in C language, and can be built and executed on top of a real–time Xenomai–based Linux system, to control real robots. The software for the I/O with the hardware has to be provided and interfaced with the motor servo, to realize a complete system as described before.

An interesting and useful feature of SL is that it can be configured to have the motor servo interact with an additional process implementing a simulator (also included in the package), instead of a real robot. The task servo, on the other hand, is unaware of the current mode of execution. This feature, which is possible thanks to the modular design described in the previous sections, is very useful since it allows to try robot behaviors first in a physics simulator and

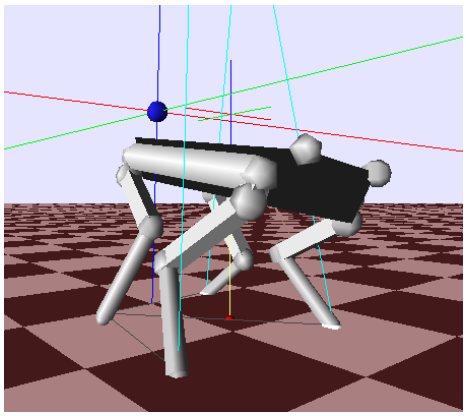


Figure 4.8 – A screenshot of the graphical output of the simulator of SL, showing the model of the HyQ robot. The intersecting lines above the robot (red, green and blue) represent the axes of the world reference frame. The light-blue lines coming from the feet of HyQ represent the ground reaction forces, while the tiny red ball on the ground is the projection of the center of mass of the robot.

then on the real robot, without changing anything in the implementation of the behavior itself.
Figure 4.8 shows a snapshot of the graphics window of SL captured during a simulation with the HyQ model.

Chapter 5

Domain analysis and software models

In this chapter we describe the domain we address in our code generation framework. We focus on a specific yet fundamental area of the robotics software which deals with model–based control and more generally with kinematics and dynamics, which are the subject of the next sections. We designed simple yet general domain models of the relevant aspects of the kinematics and dynamics of articulated robots, to then build Domain Specific Languages on top of them; the final purpose is to automatically generate code (Chapter 6 describes the actual implementation of the code generation framework).

This chapter is structured as follows: Section 5.1 gives an overview of the kinematics and the dynamics for articulated robots. This overview leads to a more rigorous analysis in view of the development of software, in the section about domain models (5.2).

5.1 Kinematics and dynamics of articulated robots

This section provides an overview of the most important algorithms and quantities related to the kinematics and the dynamics of articulated robots. It introduces the issues related to their implementation which motivate the content of the other sections.
Briefly, kinematics deals with the geometry of the robot (lengths, configuration of the links), the position and the velocity of points of interests. Dynamics considers also forces, inertia, accelerations and the physics laws that relate such quantities together. It would be wrong though, to think of these two aspects as completely independent, since the dynamics computations for a multibody system depend strongly on its geometry and thus on kinematics quantities.
These algorithms are required by model–based controllers but also by other components of the whole software system. Some of them are mandatory to realize simulations and basic controllers, making their implementation a critical step in the development of any robotics application.
For example the so called *forward kinematics* (i.e. coordinate transformation

matrices) is necessary to calculate the position of any point of the robot with respect to a reference frame, for any given configuration of the joints. Most applications require this functionality; for example one needs to know the actual position of a certain sensor (like a stereo camera) to properly interpret its measurements. More specific approaches like the operational space control are heavily based on a wide set of matrices related to kinematics and dynamics: coordinate transforms, Jacobians, inertia matrices and so on (see the related work section 3.2).

Despite an established theoretical understanding of kinematics and dynamics, sound designs and implementations – possibly meeting some of the features listed in Section 2.1 – are far from being trivial, yet they are critical for successful simulation and control and therefore for the safety of the robot itself. Writing this kind of software is error prone and time consuming, because of the inherent complexity of the algorithms but also because of the lack of established, general software models which address these concepts, and thus the lack of reusable implementations. Some algebraic quantities like the transforms typically suffer from ambiguities in their definition or in their usage (see below), so that developing them by hand is even more problematic.
An approach based on generating code out of higher level models follows naturally from (a) the nature of the dynamics and kinematics algorithms (see for example Section 5.1.2.3), (b) the issues inherent to manual development (e.g. Section 5.1.1.1), (c) the apparently conflicting requirements of efficiency and real–time capability on one hand, and ease of use and flexibility on the other hand.

5.1.1 Kinematics

5.1.1.1 Coordinate transformation matrices

A coordinate transformation matrix X (or just "coordinate transform" or "roto–translation matrix") maps a coordinate vector to another coordinate vector representing the same physical entity (e.g. a point), but in a different reference frame which is in general translated and rotated with respect to the previous one:

$$p_B = {}_B\mathbf{X}_A \cdot p_A$$

where A and B are two frames, p is the coordinate vector; more on the notation in the following.
When developing these matrices by hand or when using existing code, it is very easy to make mistakes because a proper and unique definition of a transform requires some specifications that are usually implicit and therefore lead to ambiguities. The most important one is the *direction* of the transform: as an example, assume that frame B is rotated by an angle θ about the z axis, with respect to A; this *information* cannot be misunderstood (assuming right handed frame and right hand rule for the angle sign), but the expression for the rotation matrix $R_z(\theta)$ does not tell whether it transforms vectors from B to A or vice versa, thus is ambiguous. See also Figure 5.1. If one specified all the necessary information (e.g. positive rotations are counter clockwise) and chose a convention so that, for instance, $R_z(\theta)$ must map coordinates from B (the rotated frame) to A, then one of the matrices in the figure would simply be wrong (the one on the right, in this case); in fact it would be the matrix

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 5.1 – Two rotation matrices both representing a rotation of an angle θ about the z axis ($R_z(\theta)$). One matrix is actually the inverse of the other (or, equivalently, the transpose, since rotation matrices are orthogonal). They map coordinate vectors between two reference frames which are rotated with respect to each other.

representing a rotation of $-\theta$. But the problem is exactly that this information is often missing, and the user wastes time by manually investigating about the conventions and by making mistakes. For instance, the code in the MATLAB robotics toolbox (Corke 1996) uses one convention about the direction of the transform, while the 6D transforms introduced by Featherstone in (Featherstone 2008) use the other one.

For the sake of simplicity we will focus only on the direction property, assuming to deal always with right handed, orthogonal coordinate frames and left–multiplication (i.e. the matrix always left–multiply a column vector of coordinates), these being other possible ambiguities in the usage of the transforms. This assumption is not a big limitation since different choices are quite uncommon in the literature as well as in existing software.

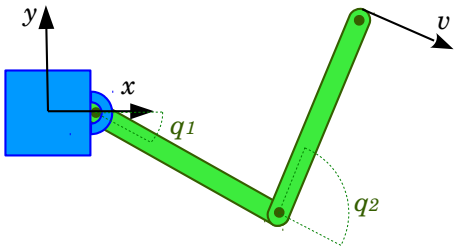
5.1.1.2 Jacobians

A Jacobian, in common robotics usage, is a matrix that maps velocities in different coordinate systems; usually a Jacobian $\mathbf{J}$ transforms the velocity vector containing all the scalar velocities of the joints (i.e. the joints–space velocity $\dot{\mathbf{q}}$) to the Cartesian velocity of some point of interest $\dot{\mathbf{x}}$ located in the robot (Figure 5.2). The well known formula is

$$\dot{\mathbf{x}} = \mathbf{J}(\mathbf{q})\ \dot{\mathbf{q}}$$

The equation also says that the matrix is a function of the current configuration $\mathbf{q}$ of the robot. Jacobians are of fundamental importance in many robotics applications since they encode effectively information about the configuration of the robot (e.g. one can determine it is approaching a kinematic singularity by looking at the Jacobian) that is also very useful for dynamics computations.

The literature usually makes a distinction between *analytical Jacobian* and *geometric Jacobian*, these being respectively the matrix of the partial derivatives of the forward kinematics function (this matches the common definition of Jacobian in mathematics) and a matrix determined by a geometric procedure. The two are in general not equal, but they are always related by a linear transformation (Khatib 1995; Siciliano et al. 2009). In this work we focus only on geometric Jacobians because they can be computed by a procedure that iterates over the joints; they also have the property of expressing the rotational velocity of the point of interest in terms of the angular velocity pseudo vector ω, and



$$\mathbf{J} = \begin{pmatrix} -\sin(q_2+q_1) - \sin q_1 & -\sin(q_2+q_1) \\ \cos(q_2+q_1) + \cos q_1 & \cos(q_2+q_1) \end{pmatrix}$$

Figure 5.2 – A simple example of a planar two–bar linkage and the corresponding Jacobian for the tip of the second bar (to simplify the results, the bars have unit length). Note that $\mathbf{J}$ is a function of the joint status variables q_1 (negative, in this example) and q_2; the equation is $\mathbf{v} = \mathbf{J} \cdot [\dot{q}_1 \ \dot{q}_2]^T$. A few rows of the Jacobian have been dropped since they contain only zeros, because motion is mechanically constrained in the xy plane.

not in terms of the derivatives of the angular position coordinates (which is the case of the analytical Jacobian).

For a description of the procedure to compute geometric Jacobians see (Siciliano et al. 2009). In short, such a procedure iteratively determines the contribution of the motion of each joint of the robot to the linear and angular velocity of the point of interest, by using information about the kinematics of the robot (e.g. the length of links, the orientation in space of the rotation axes of the joints).
This procedure can also easily lead to mistakes if followed manually, and on the other hand it lends itself well to an automated, generic implementation.

5.1.2 Dynamics

5.1.2.1 Reference equations

The very well know equation of motion for a multi–rigid–body system can be written in the following form:

$$\mathbf{\Gamma} = \mathbf{H}(\mathbf{q})\ \ddot{\mathbf{q}} \ + \ \mathbf{h}(\mathbf{q}, \dot{\mathbf{q}}) \tag{5.1}$$

where $\mathbf{q}$, $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ are respectively the vectors of the scalar position, velocity and acceleration for each degree of freedom of the robot (i.e. for each joint, assuming only 1–DoF joints), while $\mathbf{\Gamma}$ represents the forces at the joints; here and throughout the rest of this work, unless otherwise specified, we use the term "forces" to mean "generalized forces", which accounts for both the cases of linear forces or torques (respectively for prismatic or revolute joints). $\mathbf{H}$ is the matrix that expresses the inertia seen locally at each joint, while $\mathbf{h}$ is a term that accounts for the position and the velocity dependent forces; this last term – often called the "bias force" term – is in fact a place–holder for the sum of two components representing the Coriolis/Centripetal forces and gravity terms (and possibly other additional forces determined for instance by contacts, springs,

etc.). The equation can then be rewritten as:

$$\boldsymbol{\Gamma} = \mathbf{H}(\mathbf{q})\ \ddot{\mathbf{q}} \ + \ \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) \ + \ \mathbf{G}(\mathbf{q}) \tag{5.2}$$

Two main sub–problems of dynamics can be usually identified, and they are simply called the inverse dynamics problem and the forward dynamics problem. The opposite "direction" of the two refer to the opposite role played by accelerations and forces: the inverse dynamics problem aims at finding the proper forces that would lead to certain know accelerations, while a forward dynamics procedure computes the accelerations resulting from the application of known forces. We can write:

$$\boldsymbol{\Gamma} = id(\ddot{\mathbf{q}}, \mathbf{q}, \dot{\mathbf{q}}) \tag{5.3}$$

$$\ddot{\mathbf{q}} = fd(\boldsymbol{\Gamma}, \mathbf{q}, \dot{\mathbf{q}}) \tag{5.4}$$

Generally speaking, inverse dynamics is used in the controllers of real robots since they have to determine the forces to be delivered by the actuators (cf. Section 2.2). On the other hand forward dynamics is mostly used in simulators, which need to estimate the acceleration response of a multibody system in order to provide physically consistent simulations. In first approximation, on real robots the second case obviously does not apply because motion happens for real and the accelerations can be measured. However, a sophisticated control strategy of a robot (in the broad sense of the term) might very well run dynamics simulations about its own robot model, in order to *plan* or *predict* behaviors or events.
Whichever use of these routines, once again we shall stress the importance of efficient implementations. Fast and predictable execution times allow to run dynamics computations in hard real–time controllers, leaving more room for other procedures. Faster dynamics simulations improve the user experience, allow a simulator to be augmented with more complex models, and also enable sophisticated controller logic such as sampling–based planning techniques (LaValle 2006).

5.1.2.2 Algorithms

By comparing equation 5.3 with 5.1 it is easy to identify a correspondence and guess, for instance, that one can solve the inverse dynamics problem by computing $\mathbf{H}$ and $\mathbf{h}$. However this is not always the case for the dynamics algorithms. For example, the fastest know algorithm for inverse dynamics, the *Recursive Newton–Euler algorithm*, does not compute explicitly (i.e. as a side result) the coefficients of the equation of motion, since that is not the most efficient strategy to get the accelerations.

Another algorithm considered in this work is the *Composite–Rigid–Body algorithm*, which instead computes exactly $\mathbf{H}$; this matrix is known as the Joint Space Inertia Matrix (JSIM). As opposed to the inverse dynamics case, $\mathbf{H}$ is used explicitly to solve the forward dynamics problems, together with a routine to compute $\mathbf{h}$ (which is an inverse dynamics routine!) and another one to solve equation 5.1 for $\ddot{\mathbf{q}}$ – even though, depending on the characteristics of the robot, this approach might not be the fastest (Featherstone 2008).
The JSIM also appears explicitly in the formulation of operational space control and impedance control (Hogan 1985; Khatib 1995). For example it is required to

compute the *inertia weighted pseudo–inverse* of the Jacobian matrix; this inverse is extensively used in operational space control and in the task prioritization approach (Sentis et al. 2005). The corresponding equation is the following:

$$\overline{\mathbf{J}} = \mathbf{H}^{-1}\ \mathbf{J}^{T}\ \left(\mathbf{J}\ \mathbf{H}^{-1}\ \mathbf{J}^{T}\right)^{-1} \tag{5.5}$$

Another fast algorithm used to solve the forward dynamics problem is the *Articulated–Body algorithm*, which belongs to the so called "propagation methods" (Featherstone 2008); our framework can also generate an optimized implementation of this algorithm, for any kinematic tree.
Roughly speaking, all the algorithms mentioned in this section share a common behavior which is induced by the nature of multibody systems. Basically it always happens that some quantities (like forces or velocities) are computed for some body and then *propagated* to the rest of the system according to the geometry of the robot. Bodies are ordered by establishing a directed path from the robot base to its extremities (see Section 5.2.1.1). For example, for a fixed base robot, the velocity of a body B_i contributes to the velocities of all the following bodies, according to their distance from B_i (an independent contribution to a body velocity, instead, comes from the actuation at the corresponding joint). The instantaneous inertia properties of a subset of rigid–bodies connected together are determined by a weighted sum of the inertia of the single bodies, which again depend on distances and angular displacements.

5.1.2.3 Dependency on the robot model

The simple equations 5.3 and 5.4 already show the dependency on position and velocity for both inverse and forward dynamics: the instantaneous relation between forces and accelerations depend on the *kinematic configuration* of the robot.
As noted in (Featherstone 2008, 2013) though, what they do not show explicitly is the additional dependency on the *model* of the robot describing its kinematics and dynamics; such a model is required to interpret properly the other parameters (e.g. a specific force leads to very different accelerations if the links of the robot are twice as heavy, or half as long). This is a key point about the nature of dynamics algorithms which can be exploited by the code generation process. In essence, the robot model can be thought of as an additional parameter of the dynamics problems, which reveals the possibility of implementing specific *instances* of the general algorithms tailored for specific robot models.

5.2 Domain models

This section complements the previous ones with some further information regarding the kinematics/dynamics of articulated robots, in order to devise the software models – in the form of UML class diagrams, in this case – which our toolchain is based on. These models provide several advantages; they serve as a concise yet formal representation of the information we are interested in, which is what the languages and code generators described later are based on; specifically, we refer to the kinematic tree abstraction for articulated robots (Section 5.2.1) and to rigid motions (translation and rotations – Section 5.2.2). The models are not only useful for the development of the code generation system

itself, but also for its usage by third party users. In other words the models also serve as documentation, in that they can communicate the ideas and the assumptions underlying the software, making its re-use easier.

5.2.1 Kinematic trees

The previous section about dynamics introduced the concept of model–based algorithms, which are parametrized on the kinematics/dynamics description of a robot; we call such a description a robot *model*, since it is an abstraction of the actual physics of a multibody assembly. The main assumption underlying the models is that all the bodies comprising the system are perfectly rigid, hence they do not bend nor deform.
As far as dynamics is concerned, it is also assumed that joints are equipped with idealized sources of force, neglecting the dynamics of possible concrete actuation systems (like electric or hydraulic motors); the mass, the position of the center of mass and the inertia tensor of each rigid body are the sole parameters that need to be specified.
As far as kinematics is concerned, we shall give here a description of the structure and the amount of information embedded in the model, to provide the background for the rest of the work. For an extensive and authoritative discussion on the kinematic tree abstraction, which is a standard approach in robotics, see (Featherstone 2008; Siciliano et al. 2009).

5.2.1.1 Basic model

In kinematic models, a robot is an assembly of bodies, often referred to as *links*, and joints: a link is a rigid body with inertia properties while a joint is modelled as a *kinematic constraint* between exactly two bodies (the predecessor and successor), that is, it is an abstract, massless object that constrains the possible relative motion between the two bodies. In general the joint is not a purely rigid junction but it leaves certain *degrees of freedom* (DoF) to the attached link. Note that the mass of the real, mechanical joint (e.g. the extremities of two links, the bearings, the axle, etc) has to be included in the inertial properties of the connected links, or neglected. In essence, the model has to specify the type of each joint, which tells the mobility of the link and the position of the joints with respect to the bodies; this data fully describes the geometry of the robot.
The overall description of the robot is topological, since the whole structure can be represented by a graph where joints are arcs and bodies are nodes (quite the contrary of what graphical intuition might suggest); see also figure 5.3.[1]

This work focuses only on kinematic trees, which are assemblies that do not exhibit loops and represent a wide class of the robots used in industry and research; the graph abstraction is where the term “tree” comes from, since a graph with no loops is called tree. The full generalization of the model is one of the natural topics for future development, and can be done by integrating the methods described again in (Featherstone 2008) into our framework.
As a further simplification, we deal only with 1–DoF joints, that is joints that

[1] This abstraction suggests how graph–related data structures and algorithms may be used in computer programs to implement the kinematic tree model.

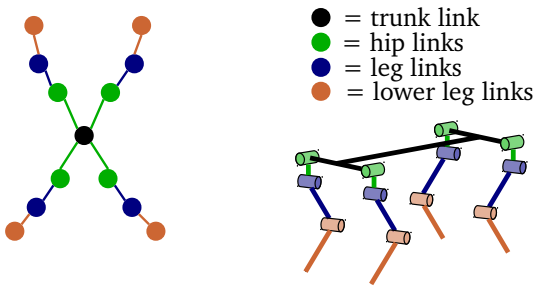


Figure 5.3 – The structure of articulated robots can be modelled with a graph where the rigid links are the nodes and the joints the arcs. This example shows a graph isomorphic to the kinematic structure of the quadruped robot HyQ, also shown on the left with a stylized drawing, for reference.

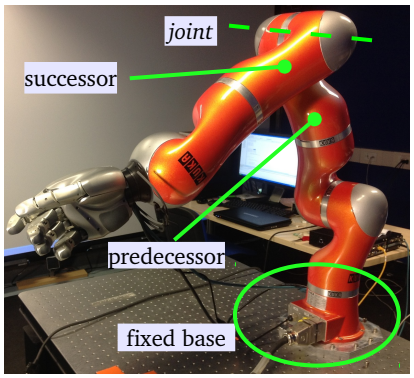
guarantee only rotational motion about one axis (revolute joints) or linear motion along one axis (prismatic joints). More complex mechanisms can still be modelled by multiple simpler joints, which therefore can be thought of as virtual joints; this approach is convenient since it does not affect kinematics and dynamics routines. In certain situations though, more complex and dedicated models would be needed to cope with specific issues; for instance the straightforward modeling of a 3-DoF spherical joint as a sequence of three revolute joints would possibly suffer from the well know singularities related to the parametrization of rotations with three numbers (Featherstone 2008; Siciliano et al. 2009; Stuelpnagel 1964). This point will be addressed in future developments of this work.

One particular link of the robot is called the robot *base*, which is the logical root of the kinematic tree. A robot anchored to the environment, like an industrial manipulator, is called *fixed–base* and its actual base is the link attached to the environment; such link does not move thus is neglected in any dynamics computation.

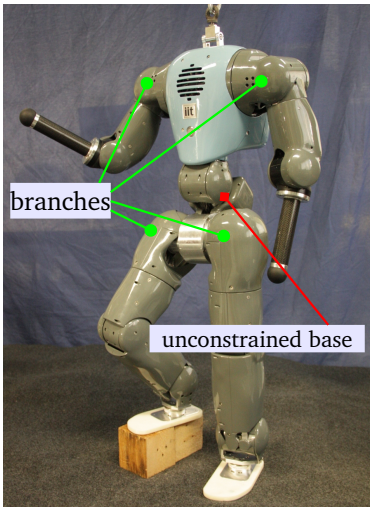
Robots like humanoids or quadrupeds, on the other hand, are called *floating base* since the whole robot position is not constrained somewhere in the environment; in this case the choice of the base link is arbitrary in principle, even though some guidelines may be applied. For example a typical choice for a legged robot is the trunk, since it matches our intuition and minimize the depth of the branches (legs and arms), which can be helpful for the efficiency of certain algorithms.

The choice of the base also determines the logical hierarchy among the links, that is which link is the parent of which other link. It also influences the numbering of joints and links, even though in case of branches (see below) the user still has some freedom in assigning the numbers (details about these points are not relevant here, see (Featherstone 2008)). We shall give here a few more definitions related to kinematic trees that will be used in the rest of the text; compare them also with Figure 5.4. Note that they apply to the simplified case that assumes no kinematic loops and neglects the joint polarity property; for a more exhaustive description of the kinematic model of rigid body systems please refer to chapter 4 of (Featherstone 2008).

- When referring to a link, we call “supporting joint” the joint that moves



(a) A fixed-base manipulator arm with a linear structure (the KUKA–DLR robot arm (Bischoff, Kurth, et al. 2010)). The highlighted predecessor-successor pair refer to the joint whose axis is highlighted.



(b) A floating base robot with kinematic branches (the humanoid robot COMAN (Tsagarakis et al. 2013)).

Figure 5.4 – Two examples of articulated robots – fixed and floating base – illustrating some of the main elements of the kinematic tree model.

such a link (and thus the whole sub–tree rooted at it – for example the elbow is the supporting joint of the lower arm); when referring to a joint, we also call "supporting link" the link that is carrying that joint (the upper arm with respect to the elbow);

- For any section of the tree in the form link P - joint J - link S, where P is the link supporting joint J and S is the link supported in turn by J, we call P and S respectively the *predecessor* and the *successor* of J. The same two links may be equivalently referred to as the *parent* and the *child* link.

- Kinematic branches or simply branches are forks in the structure of the robot that happen when a link is supporting two or more joints (which are in turn supporting other links). For example, the trunk of a quadruped robot is supporting the four joints that act as hips and shoulders.

- A kinematic chain is a linear sequence of links connected through joints, with no branches. An arm of a humanoid robot up to the wrist (neglecting a possible hand) is an example of a chain.

5.2.1.2 Reference frames

The geometry of the bodies and their connections is required to dynamically compute the *pose* of the bodies, the dynamical effects of the movements, such as Coriolis and centrifugal forces, and so on. To this end, various reference frames must be placed in known points of every body and every joint of the tree. The parameters for a set of *transformations* among different frames *plus*

a convention about the placement of them (e.g. the z axis of the frame of a joint is always aligned with the rotation axis) basically encode all the required geometric information. Figure 5.5 shows the layout of the reference frames on

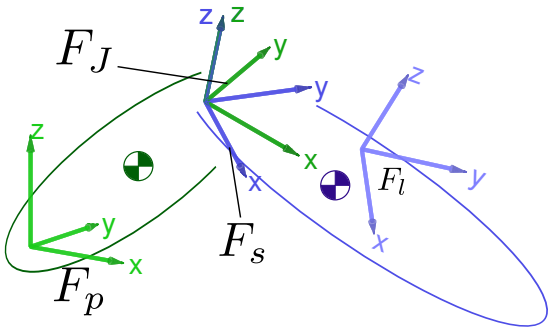


Figure 5.5 – Layout of reference frames for a generic section of a kinematic chain. F_p and F_s are the frames respectively of the predecessor and successor link of joint J, whose frame is F_J. F_p and F_J do not move with respect to each other, while F_J and F_s do, according to the joint behavior. F_l shows a possible additional frame located on the link.
For further information about the convention please refer to (Featherstone 2008, 2010b)

.

a generic section of a kinematic chain. We shall emphasize here some of the main characteristics of the convention about the placement of frames which are relevant for the overall understanding of the model and hence of the language described in Section 6.1.

- The joint frame F_J is always placed with the z axis aligned with the actual joint axis, either a rotation axis or a translation one. This way, the placement of F_J fully specifies the motion allowed by the joint.
- The transform ${}_J\mathbf{X}_p$ for the joint frame is a *constant*, since it describes the placement of F_J (thus of the joint itself – see the previous point) expressed in the reference of the supporting link (the predecessor); in other words ${}_J\mathbf{X}_p$ solely depends on static, geometrical parameters of the robot (usually included in its documentation papers), which can be completely arbitrary.
- Two reference frames are located at each joint (F_J and F_s), and they are fully overlapped when the joint status (i.e. the actual angle or displacement) is zero. Only the second frame (F_s) moves as the joint is actuated, as it is attached to the successor link. This frame is chosen to be the default frame of such a link.
- From the previous points, it follows that no additional parameters have to be specified for the link, since its reference frame is uniquely determined by the convention about the placement of the joint frames, and the joint status.
- The generic transform ${}_s\mathbf{X}_J$ between the two frames at the joint (F_J and F_s) is the only one which depends on the joint status. Note also that ${}_s\mathbf{X}_J$ captures the *type* of the joint, that is whether it is revolute or prismatic.

Note that the procedure described by the above points poses no constraint on the actual shape or layout of the mechanical system to be modeled. It also does not constrain the user to a limited set of predefined frames that may not fit her or his needs, since additional, arbitrary reference frames can be added as will (see Section 6.1).

5.2.1.3 Software model

Figure 5.6 shows an UML class diagram representing the key elements described in the previous paragraphs. Class diagrams are the typical choice when the objects of a system and the static relationships among them have to be represented (Fowler 2003). In our case we want to formalize the description of our domain of interest, which is the representation of robots as kinematic trees, which is common in robotics.
Before moving further with the explanation it is worth spending some words to avoid confusion about the terminology. The representation of articulated robots as a tree–like assembly of links and joints, is itself referred to as the "kinematic *model*"; however, a specific description of a single robot is often called the robot *model* as well. The same term is used for two things that are conceptually very different: the latter refers to the description of a specific robot, while the former deals with how the descriptions themselves are structured. To be more precise then, one might refer to the kinematic tree model as a *meta–model*, since it does not address directly robots but rather the way they can be represented. Robot–models shall then be thought of as *instances* of the meta–model. However, the distinction is not always fundamental, but it becomes relevant when there is the need to differentiate between the two cases.
The UML class diagram can also be called a model, or a domain model; to avoid further confusion, it can just be thought of as a graphical representation of the kinematic tree meta–model that has been described in words before. Such a representation is more compact and formal, thus more appropriate to serve as a reference for software development.

The class diagram is simple but general, and can be applied to any robot made by rigid links, provided that no kinematic loops are present. As expected, the main classes are `Link` and `Joint`. Since tree–joints induce a parent–child relationship among links, which is further specified by the type of joint, we chose to represent this feature by making `Joint` an association class connected to the self–association for `Link`.
Any link can have multiple children (but only one parent), which allows to represent kinematic branches such as multiples legs attached to a single trunk link in a legged robot. On the other hand kinematic loops are not addressed in this class diagram: another type of joint not determining any new child link but rather connecting two existing links would be required. Kinematics loops have not been addressed in this work mainly because the dynamics algorithms for such a class of mechanisms are much more involved (however, support for closed loop systems may be added in the future including the methods that are available in the literature).
The abstract class `Link` actually models any rigid body with inertia properties, and has a few subclasses to differentiate among special cases:

- `ChainLink`: a generic piece of the kinematic chain; this is what is usually referred to as link.

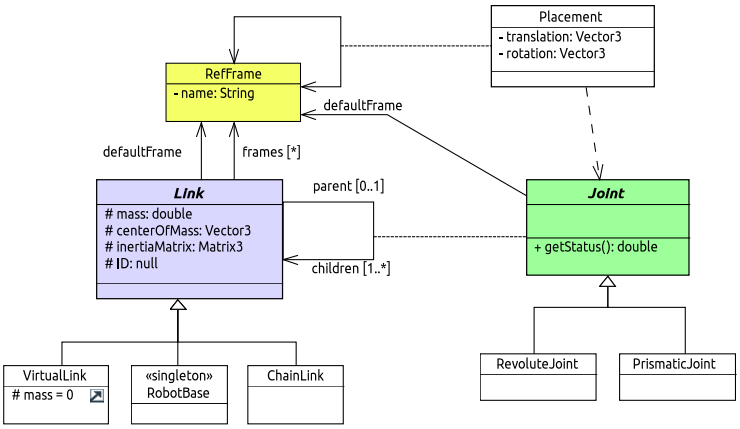


Figure 5.6 – The kinematic tree (meta–)model as an UML class diagram. It can represent multibody systems, capturing the role of a joint as the connection between a parent-child pair of bodies (links), and emphasizing their association with reference frames. The convention about the placement of these frames (that cannot be modeled in the diagram) and the parameters about the relative pose of two successive ones (the `Placement` class) provide the full geometry information about the robot.

- `RobotBase`: a special link which represents the root of the kinematic tree. It can be floating if the robot is mobile, such as a locomoting robot. The stereotype `Singleton` means that only one instance of this class shall exist for each robot model, since any robot has only one base.

- `VirtualLink`: a dimensionless body to allow the composition of primitive joints to model more complex mechanisms (see Section 5.2.1.1); this class explicitly forces the inertia parameters of its instances to be zero. Floating base robots can be thought of as connected via a virtual six-DoF joint (i.e. no constraint), to an arbitrary point in the world (cf. (Featherstone 2008)).

The `Joint` class is also sub-classed to distinguish between prismatic and revolute joints. The common operation `getStatus()` shows that it must be possible in some way to inspect the joint variable value (a scalar, since we are dealing with 1-DoF only joints) about the current position of the joint. The vector containing all the scalar values at a specific instant is the joint–space state variable, commonly called $\mathbf{q}$.

Finally, the diagram of Figure 5.6 also includes reference frames with the class `RefFrame`, associated with both `Link` and `Joint`; in addition to the property `defaultFrame` that refers to the main reference frames located at links and joints according to the convention, links can have any number of additional frames as required by the user. Note that since a frame per se does not really have any property nor behavior (we assume all to be right–handed), the relevant information is in fact encoded in `Placement`, which contains the positioning parameters of a frame with respect to another one.

5.2.2 Rigid body motions

The other problems we are interested in are coordinate transformation matrices and geometric Jacobians. The kinematic tree model already exposes the concepts of reference frames and relative pose of two frames, as they are required to specify the geometry of the robot. Coordinate transformations though, represent an independent problem that as such deserves separate treatment; as described later, this separation improves the re–usability of the code generator facilities.
As we did for kinematic trees, we shall devise some sort of software model that can point out the main objects and relationships of the domain, to serve as reference for software development (in our specific case the development of a Domain Specific Language and code generators). As introduced before (Section 5.1.1.1), coordinate transforms suffer from some ambiguities that make their manual development and usage prone to errors. One of the purposes of devising formal models is to tackle such issues.

It turns out that the actual foundation of the coordinate transforms problem are rigid body motions, that is translations and rotations of rigid bodies (or of reference frames, which as far as we are concerned is the same thing). Figure 5.7 contains another UML class diagram concerning rigid body motions. It formalizes the information that can uniquely express any motion, thought of as the sequence of movements to be applied to a reference frame to bring it in the same pose of another frame.

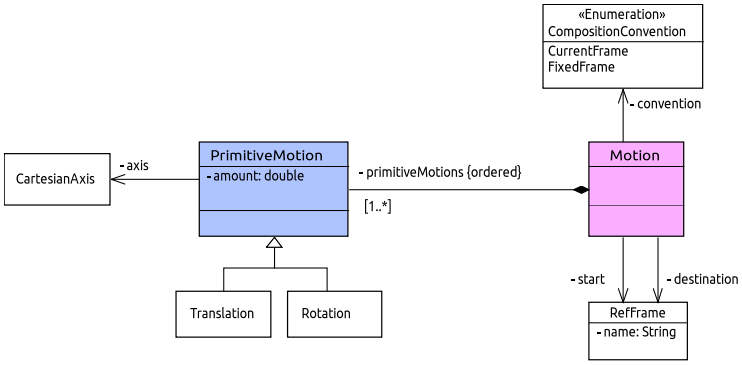


Figure 5.7 – A UML class diagram representing rigid motions. A rigid motion (`Motion`) is a sequence of movements, rotations and translations, a generic body has to perform to go from one pose to another pose; poses are represented by reference frames. The interpretation of the ordered sequence of primitive motions constituting a composite one, is unique because of the `convention` property.

The class `PrimitiveMotion` represents either a pure rotation or a pure translation of arbitrary amount, about or along a single Cartesian axis; instances of this class are basically members of one–dimensional subgroups of $SO(3)$ (the group of special orthogonal matrices that represent arbitrary pure rotations) and of $\mathbb{R}^3$ (arbitrary translations). A composite motion – class `Motion` – is basically an *ordered sequence* of primitive motions, since by composing simple rotations and translations one can achieve any motion.

Note that this information is not yet enough to uniquely determine a motion: it must be specified with respect to what reference the primitive motions (also called "motion steps") have to be taken. The natural choice is to assume that each motion step is performed with respect to the current moving frame, so that, for instance, a simple rotation about the y axis happens about the y axis of the moving frame in its current pose (i.e. after all the previous motion steps). However, one might want all the motion steps to be expressed with respect to the frame in the initial position, so that the reference axes for translations and rotations never move. Note that in this case the order of the primitive translations does not matter. Although even more conventions could be adopted, for simplicity and because of the low probability of encountering them, we shall stick with these two; the point is more about the need of explicitly exposing properties that can resolve ambiguities of this kind, especially if software dealing with these objects has to be developed. In the diagram of Figure 5.7, the property `convention` refers exactly to this point.

At this point an *instance* of the model represented by the class diagram contains all the required information to identify uniquely a set of rigid body motions, which is the foundation to develop coordinate transformation matrices.

In mathematics, the group of rigid motions is denoted with $SE(3)$, the Special Euclidean group, which is also a Lie group of dimension 6 (i.e. an unconstrained rigid body in space has six degrees of freedom). There exist different *representations* of $SE(3)$, which are defined as linear maps whose image is one of the general linear groups $GL(n)$ (whose elements are $n \times n$ matrices) (Selig 2005). One can indeed think of a coordinate transform as a representation of a rigid motion; more than one of such representations are possible, but what we are mostly interested in here are homogeneous coordinate transforms (that are 4×4) and spatial vector transforms (that are 6×6) (Featherstone 2010a,b). As we will see, this idea of different representations of the same information turns out to be very useful for code generation, because code for different kind of matrices can be generated given the same input, significantly easing the work of the developer.

5.2.2.1 Coordinate transforms

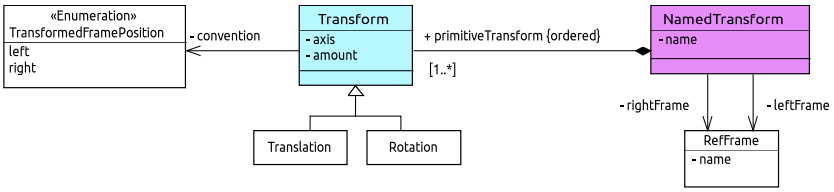


Figure 5.8 – The UML class diagram for coordinate transforms. A generic, named transformation matrix results from the product of several primitive transforms, which correspond to pure rotations or translations. The `convention` property allows to determine uniquely the elements of a primitive transform.

Figure 5.8 shows a class diagram for coordinate transformation matrices. As expected, this diagram resembles quite closely the previous one about motions,

since there is a tight correlation between motion and coordinate transforms, even though the two domains are not the same thing. The diagram is roughly a model of the possible representations of rigid body motions, that is, generic coordinate transforms whose actual type (homogeneous transforms, spatial vector transforms, or even quaternions if only the rotation part is of interest) is unspecified.
Thus, one can think of instances of the main classes `PrimitiveTransform` and `Transform` as $n \times n$ matrices, which can be composed by means of the regular matrix product. It is assumed that the matrix is always left multiplying a column vector of coordinates[2], so that it transforms the coordinates on its right to the coordinates on its left, as in the following notation (already used in Section 5.1.1.1):

$$p_B = {}_B\mathbf{X}_A \cdot p_A$$

In the remainder of the text, we use the terms *left frame* and *right frame* referring to the position of the subscripts of $\mathbf{X}$ and therefore to the frames involved in the transform (respectively B and A in the example): the *right frame* is the frame in which the coordinates being multiplied are expressed; the *left frame* is the frame in which the resulting coordinates are expressed.

Some details about the classes follow:

- `PrimitiveTransform` represents plain coordinate transformation matrices associated to primitive motions (translations or rotations involving one Cartesian axis only). The `convention` property serves to avoid the ambiguity described before in Section 5.1.1.1 and in Figure 5.1. It tells whether the frame $\mathfrak{F}_M$ that is moved with respect to the starting one $\mathfrak{F}_S$ (for example by a rotation of θ radians about the z axis) is the right or the left frame for the matrix; this information determines uniquely its elements. Specifically, if the property value is *right*, then the matrix maps coordinates expressed in the moved frame to coordinates in the original frame, i.e. it takes this form: ${}_S\mathbf{X}_M$; in the other case it is simply the other way round.

- The class `Transform` is defined as an ordered sequence of simple transforms (i.e. a product of matrices), which can also have a user specific name. The important attributes `leftFrame` and `rightFrame` identify uniquely the role of the matrix by stating explicitly which are the frames whose coordinates are involved in the transformation, and the *direction* of the transformation itself (see Section 5.1.1.1). As explained above already, the transform is supposed to multiply a vector of coordinates in the frame `rightFrame` and give as a result the coordinates in the frame `leftFrame`.

It is clear that also in this case a relevant part of the diagram lies in the properties that ensure that any instance of the model (i.e. some data structure representing coordinate transforms) is uniquely interpretable. This is fundamental in order to make a software capable of dealing robustly and consistently

[2]This is almost always the case in research literature and books. It would be possible to avoid the assumption and make the model and the software even more flexible by adding one more property and additional logic. However we decided to avoid it for the sake of simplicity and because the other convention (right multiplying) is quite rare. Similar comments apply for the case of left handed coordinate frames (see Section 5.1.1.1).

with these objects; for example, a code generator that can emit the code implementing the correct transformation matrix given the sequence of the simple transforms it is composed of.

5.2.2.2 Jacobians

As for a coordinate transform, also a geometric Jacobian is uniquely identified by two reference frames, which we call the "target frame" and the "origin frame". They are respectively the frame of the body whose velocity (or force) is of interest and the frame in which the velocity (or force) will be expressed.[3] In the robotics literature it is common to find expressions like "end-effector Jacobian" or "constraint Jacobian", where these expressions are really just shortcuts to identify the frame of interest, without mentioning the origin one, since it is assumed to be known from the context (and often coincides with the robot base frame). However, to have software deal automatically with these objects in a general way, this information cannot be left implicit.

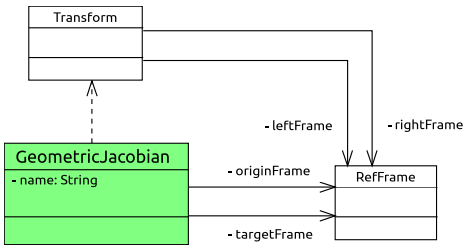


Figure 5.9 – The class diagram of Figure 5.8 with the addition of class to model geometric Jacobians.

The domain model including Jacobians is shown in Fig. 5.9. The diagram shows, by means of a dashed arrow – as in the UML specification –, the dependency with the `NamedTransform`, since any geometric Jacobian can be fully computed on the basis of direct kinematics (Siciliano et al. 2009). Since this dependency exists as an algorithm, it cannot be detailed in this diagram that captures structural features. The diagram is indeed quite straightforward and shows only the explicit link with the reference frame class, to model the target and the origin frame of Jacobians.

[3] Actually Jacobians depend on the origin frame and on one *point* somewhere else in the multibody structure; the orientation of a possible frame in such a point is not relevant and can be set to any value. We did not introduce an additional element in the design for simplicity, and because a reference frame object can be easily used instead of a point.

Chapter 6

Implementation of the code generation framework

This chapter describes the actual toolchain we developed to create a code generator for the kinematics and the dynamics of articulated robots. Section 6.1 describes the design of three Domain Specific Languages for the specification of the kinematic model of a robot, for rigid motions and for coordinate transforms. Section 6.2 deals more specifically with the actual code generation process implemented within the languages infrastructure.

The technologies we used for the implementation of the framework include primarily the Xtext workbench for Eclipse (Eysholdt et al. 2010), which is specifically designed to support the development of custom DSLs. It provides a specification language (that happens to be a DSL itself) to create grammars that should be easier to read and understand if compared to conventional methods (such as the Backus–Naur Form BNF technique) (Efftinge et al. 2013).
The infrastructure of a DSL created by Xtext supports the transformation of the models (defined by documents) into text, i.e. code generation. The actual code generation logic must be implemented in Java or Xtend; Xtend is a high level language that compiles into Java and, among other things, provides good support for string concatenation, that is useful for code generation (Efftinge and Zarnekow 2013). We also use the symbolic engine MAXIMA (Maxima 2011), whose role within the framework is described in Section 6.2.2.3.
All these tools are open source and work with a variety of platforms, and allowed us to implement our approach without any commercial technology.

6.1 The specification languages

This section provides a detailed description of the languages we designed on top of the domain models described in Section 5.2. The languages allow the user to specify conveniently the input information for the code generation, as for instance the kinematic description of a robot.

6.1.1 Overview

The introduction about DSLs in Section 2.3, the analysis of the problems domain of Chapter 5, and the general aim of relieving the user from manual development, provide a good evidence that the DSL technology is suitable for our requirements. We can design languages that allow the user to easily specify instances of the domain models we presented in Chapter 5, e.g. to describe the kinematic model of an articulated robot, or a set of coordinate transforms of interest. These documents can be parsed, checked for semantics constraints and transformed, primarily into executable computer code such as an inverse dynamics algorithm tailored for a particular robot. The end user only deals with a high level, simple facade (i.e. the languages themselves, and the documents) over such a machinery, and the simple models he or she has to specify are decoupled from the coding, which is more complex and can be partially automated.

We chose external DSLs, whose documents can be plain text files, with a clear aspect (syntax) and intuitive semantics. This choice was almost straightforward also because of the good technological support provided by Xtext; it relieves the user from manually developing the language parser and the rest of the infrastructure, which are among the main drawbacks of external languages.

In general, DSLs happen to be a very effective implementation technology for our framework; part of the contribution of the present work in the field of robotics software indeed lies in the investigation and in the exploitation of such a technology within the development process of software for articulated robots.

6.1.2 General features of the grammars

The core of an external DSL is the language grammar, which is a set of formal specifications concerning the syntax of the language: which are the allowed keywords, the allowed statements, the punctuation, and so on.
Basically the grammar indirectly defines the content of any possible document, and its structure. For this reason the grammar happens to be naturally coupled with the domain model of the language (Section 2.3), which in turn defines the structure of the *information* to be carried in the documents. This concept is explained in Figure 6.1, and it was applied for the development of the grammars presented in the following sections. In general, after the model is reasonably established the design of the grammar is quite straightforward, as the required effort is limited and secondary with respect to a sound understanding of the domain. This confirms the nature of DSLs as a *thin* layer on top of a model of the semantics.
Obviously the grammars of the DSLs also provide additional syntax elements that do not have a direct counterpart in the domain model, but are required to improve readability and make the language usable.

The following sections will show the details of the grammars and examples of documents to give the reader an idea about the actual design of the languages.

6.1.3 The kinematics DSL

The main language of our framework, which is the foundation of the code generation, is the language that allows to describe the kinematics (and dynamics) model of an articulated robot, according to the convention described in Section

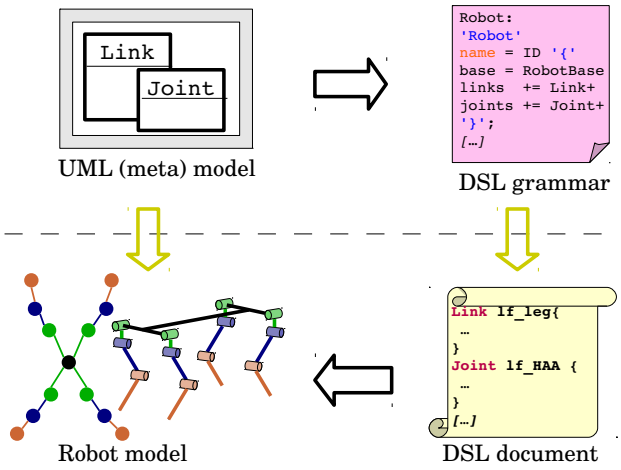


Figure 6.1 – The conceptual relationship between the domain model and the grammar of an external DSL, as well as the role of the instances of the two. Shortly, both the grammar and the domain model lie in the meta–level since they are both specifications of the general structure of actual items of the domain. Any instance document of the grammar uniquely specifies how to instantiate the model (e.g. the actual kinematics of a specific robot). Note that on the other hand the grammar does not specify the model, but it is based on it; different grammars for the same model may be devised.

5.2.1. We call it just the "kinematics DSL", for brevity.
Using this domain specific language, users can easily encode all the relevant kinematic and dynamic information about their own robots, in a compact and readable format that can be later used as input for the code generator.

6.1.3.1 The grammar

As explained before, the grammar resembles quite closely the domain model, and basically each class of the UML diagram has a corresponding element in the grammar that in turn results in some text in the documents. See Figure 6.2; the grammar is reasonably simple and even an inexperienced reader may guess a rough meaning, also with the help of Figure 6.3 that shows an example document.

The root of the document is the `Robot` element, which has a name and includes a base, links and joints. The class hierarchy for links is reflected in the grammar, that defines keywords for the robot base (that can be floating or fixed) and the regular links of the structure, all with inertia parameters.
An important part of the grammar are the rules that allow to express the actual graph–like structure of the robot: this is achieved by inserting the element `childrenList` in the `Link` elements, which expands to a possibly null list of joint-link pairs, one for each child link and the corresponding supporting joint.
General syntax features include the use of curly braces to enclose sections of text (as the set of properties of a link), which is a common feature taken from programming languages that however has an intuitive meaning even for non programmers. Other features include the syntax `<key> = <value>` which is

```
Robot:
    'Robot' name = ID '{'
    base = RobotBase
    links  += Link+
    joints += Joint+
    '}';

AbstractLink:
    Link | RobotBase;

RobotBase:
    FixedRobotBase | FloatingRobotBase;

FixedRobotBase:
    'RobotBase' name=ID
    '{'
    inertiaParams = InertiaParams
    childrenList = ChildrenList
    ('frames' ('{'frames += RefFrame+'}')+)?
    '}';

FloatingRobotBase:
    'RobotBase' name=ID 'floating'
   '{'
    inertiaParams = InertiaParams
    childrenList = ChildrenList
    ('frames' ('{'frames += RefFrame+'}')+)?
    '}';

Link:
    'link' name=ID '{'
    'id' '=' num = MY_ID
    inertiaParams = InertiaParams
    childrenList = ChildrenList
    ('frames' ('{'frames += RefFrame+'}')+)?
    '}';

InertiaParams:
    'inertia_params' '{'
        'mass' '=' mass=FLOAT
        'CoM'  '=' com= Vector3
        ('Ix''=' ix  = FLOAT & 'Iy''=' iy  = FLOAT &
         'Iz''=' iz  = FLOAT & 'Ixy''=' ixy = FLOAT &
         'Ixz''=' ixz = FLOAT & 'Iyz''=' iyz = FLOAT)
        ('ref_frame' '=' frame=[RefFrame])?
    '}'
;
```

[continue]

[continue]

```
Joint:
    RevoluteJoint | PrismaticJoint;

RevoluteJoint:
    'r_joint' name=ID '{'
    ('id' '=' num = MY_ID)?
    'ref_frame' '{'refFrame = RotoTrasl'}'
    '}';

PrismaticJoint:
    'p_joint' name=ID '{'
    ('id' '=' num = MY_ID)?
    'ref_frame' '{'refFrame = RotoTrasl'}'
    '}';

ChildrenList: {ChildrenList}
    'children' '{'
      (children += ChildSpec)*
    '}';

ChildSpec :
    link=[Link] 'via' joint=[Joint];

RotoTrasl:
    'translation' '=' translation = Vector3
    'rotation' '=' rotation = Vector3
    ;

RefFrame:
    name = ID '{'
    transform = RotoTrasl
    '}'
;
Vector3  :
    '('x=Var','y=Var','z=Var')';
```

Figure 6.2 – Almost the entire Xtext grammar for the kinematics DSL– at the most recent version at the time of writing –, which reflects the domain model shown in Figure 5.6. Details about the definition of the element `var` have been omitted for brevity (basically its grammar rule matches floating point numbers or simple expressions like $-\pi/2$). Every constant text enclosed in quotes (') is a keyword of the language that will appear in the documents as it is here in the grammar; upper case literals (e.g. `ID` or `FLOAT`) are terminal rules (i.e. they expand to terminal symbols and do not trigger any further rule); literals starting with a capital letters are parser rules for compound elements (e.g. `PrismaticJoint`), while lower case literals are properties (called *features* in the Xtext jargon) of such elements (e.g. `name` and `refFrame`). For more information and a complete reference, see the online documentation of Xtext (Efftinge et al. 2013).

straightforward as well; it is used for instance to allow the insertion of the values of inertia parameters.

6.1.3.2 Documents

Figure 6.3 shows an example document compliant with the proposed grammar, which allows us to further explain the features of the language. The format is very simple, neat and intuitive, yet it includes all the information to fully specify the physics of the system according to the rigid body dynamics model. All the constants of any document are implicitly expressed in SI units (e.g. inertia moments in $[kg\ m^2]$, rotations in $[rad]$, etc.).[1] Other relevant points are the following:

- The part of the text describing the floating base link (the `trunk`) is an example of how to model a branched structure, which simply boils down to inserting more than one item in the `children` sub–block. Each of those items is in the form `<link name> via <joint name>`, and together they specify all the parent–child relationships of the kinematics model (see Section 5.2.1.1).

- The numerical parameters in the blocks of the joints specify the pose of the joint frame with respect to the default frame of the supporting link; these numbers are a fundamental part of the specification of the actual kinematics of the robot. Six values are required: the first three are interpreted as a translation vector (field `translation`), the others as successive rotations about the x, y and z axis, in this order, each one expressed with respect to the current frame.

- The lower–leg link shows another important feature of the language that allows to add an arbitrary amount of additional reference frames to any link of the robot, defined with respect to the default frame (in this case the frame is only one, called `foot`, making the description more flexible. As an example, the user might want to mark in the model the position and the orientation of specific sensors or simply points of interests in the structure, especially since coordinate transforms for these frames can be generated automatically (see Section 6.2.2).

- Inertia parameters have to be inserted as numerical constants. The six distinct moments of inertia of each rigid body can be entered in any order, and there is a distinct keyword for each of them, i.e. the user does not have to enter a ordered sequence of numbers; this may seem a little detail, but prevents very common mistakes and waste of time. Note also that the values are the moments of inertia and *not* the elements of the 3×3 inertia tensor, even if the difference is just a sign.

[1] A proper modelling of physical quantities and units of measure is a fundamental issue in software – not only for the robotics domain –, as it is a major source of numerical errors and human mistakes. A proper solution to such an issue is beyond the scope of this work, thus relying implicitly on the most acknowledged standard about units is the best available option.

```
Robot HyQ {
RobotBase trunk floating {
    inertia_params {
        mass =  47.376
        CoM = (-0.0223,-0.0001, 0.0387)
        Ix  = 1.209488
        Iy  = 5.583700
        Iz  = 6.056973
        Ixy = 0.005710
        Ixz  = -0.190812
        Iyz  = -0.012668
    }
    children {
        LF_hip via LF_HAA
        RF_hip via RF_HAA
        LH_hip via LH_HAA
        RH_hip via RH_HAA
    }
}

link LF_hip {
    id = 1
    inertia_params {
        mass = 2.93
        CoM = (0.04263, 0.0, 0.16931)
        Ix  = 0.134705
        Iy  = 0.144171
        Iz  = 0.011033
        Ixy = 0.000036
        Ixz  = 0.022734
        Iyz  = 0.000051
        }
    children {
        LF_leg via LF_HFE
    }
}

link LF_leg {
    id = 2
    inertia_params {
        mass = 2.638
        CoM = (0.15074, -0.02625, -0.0)
        Ix  = 0.005495
        Iy  = 0.087136
        Iz  = 0.089871
        Ixy = -0.007418
        Ixz  = -0.000102
        Iyz  = -0.000021
    }
    children {
        LF_lowerleg via LF_KFE
    }
}
```

[continue]

[continue]

```
link LF_lowerleg {
    id = 3
    inertia_params {
        mass = 0.881
        CoM = (0.1254, 0.0005, -0.0001)
        Ix  = 0.000468
        Iy  = 0.026409
        Iz  = 0.026181
        Ixy = 0.0
        Ixz = 0.0
        Iyz = 0.0
    }
    children {}
    frames {
        LF_foot {
            translation = (0.33, 0.0, 0.0)
            rotation = (PI/2.0, -PI/2.0, 0.0)
        }
    }
}
```

[...]

```
r_joint LF_HAA {
    ref_frame {
        translation = (0.3735, 0.207, 0.0)
        rotation = (0.0, -PI/2.0, PI)
    }
}
r_joint LF_HFE {
    ref_frame {
        translation = (0.08, 0.0, 0.0)
        rotation =  (PI/2.0, 0.0, 0.0)
    }
}
r_joint LF_KFE {
    ref_frame {
        translation = (0.35, 0.0, 0.0)
        rotation =  (0.0, 0.0, 0.0)
    }
}
```

[...]

```
} // EOF
```

Figure 6.3 – An excerpt from an instance document of the kinematics DSL, modeling the quadruped robot HyQ (Semini, Tsagarakis, et al. 2011). It shows the trunk link (which is also the floating base of the robot) and the parts of one of the four legs, since the others are almost identical. `LF` stands for left-front, `RH` for right-hind, and so on; `HAA` stands for Hip-Abduction-Adduction, `HFE` for Hip-Flexion-Extension and `KFE` for Knee-Flexion-Extension. The lower-leg link defines an additional reference frame with origin in the foot of the leg. Compare the model with the grammar of the language shown in Figure 6.2.

6.1.4 The rigid body motions DSL

The purpose of this language is to provide a tool to the end user for uniquely specifying rigid body motions; for example to express that the reference frame in the shoulder of a humanoid robot is translated with a certain vector and rotated about two axes with respect to the reference frame of the trunk. As introduced in Section 2.4, the aim of this language together with the one presented in the next section, is to provide a toolchain to do robust code generation of arbitrary coordinate transforms (and also geometric Jacobians).
This language and the following one are completely independent from the previous one about kinematics tree, and in principle they can be used as standalone tools by the end users. It follows that they can also be put together, enabling the development of a rich code generation framework for robots; that is indeed what we have done, as detailed in Section 6.2.3.
We will refer to this language throughout the text as the "motion DSL".

Figure 6.4 shows the code of the grammar, which also in this case replicates closely the structure of the domain model, while Figure 6.5 gives an example of a document of the language.
The generic document starts with a name, followed by a list of declarations of the reference frames that will be mentioned in the text; reference frames are just named entities, so the list is really just a set of identifiers. For brevity, the convention about the composition of primitive motions is written only once in the preamble of the document, so that each declared motion is assumed to follow that convention.
Motions are identified by a pair of reference frames that correspond to the initial and final pose of the imaginary reference frame that is moving, according to the list of rotations and translations. The tokens for primitive motions are self-explanatory; their argument can be a floating point number, an identifier or a simple expression like 2π.

6.1.5 The coordinate transforms DSL

This language was designed to let the user write a list of abstract coordinate transformation matrices, defined as sequences of simple transforms. The idea is that code implementing various realizations of these transforms – like homogeneous transforms and spatial vectors transforms – can then be generated automatically starting from the same document.
As you can see in Figures 6.6 and 6.7, the grammar and the resulting documents are very similar to the ones of the motion DSL, although the same differences in the corresponding models still hold here in the languages.

As before, a document contains a name, a declaration of the frames referenced afterwards and a document–wide property, `TransformedFramePos`, explained below. The property is followed by a list of the user–defined transformations written with a syntax that clearly identifies the left and right frame attributes ({B}_X_{A} as in the notation ${}_B\mathbf{X}_A$), even though an optional customized name can be added at the end of the line, within square brackets.
Tokens in the form `Rx()`, `Ty()`, ..., are keywords of the language and represent the atomic transformations. The possibility of using custom identifiers as their arguments is useful to show the dependency on some kind of variable, for example the identifier for the status of a joint. The same identifier is possibly

```
Model:
    "Model" name=ID
    framesList = FramesList
    "Convention""="convention=("local"|"fixed")
    motions += Motion*
;

FramesList:
    "Frames""{"
        items += Frame
        (','items += Frame)*
     "}"
;

Frame: name=ID;

Motion:
    start=[Frame] '->' end=[Frame]
    ':' primitiveMotions += PrimitiveMotion*
    ('['userName=ID']')?
;
```

```
PrimitiveMotion:
    Rotation | Translation
;
Rotation:    Rotx | Roty | Rotz ;
Translation: Trx | Try | Trz ;

Trx:  "trx""("arg=ArgSpec")";
Try:  "try""("arg=ArgSpec")";
Trz:  "trz""("arg=ArgSpec")";

Rotx:  "rotx""("arg=ArgSpec")";
Roty:  "roty""("arg=ArgSpec")";
Rotz:  "rotz""("arg=ArgSpec")";

ArgSpec :
    FloatLiteral | Expr
;
```

Figure 6.4 – The grammar of the DSL for rigid motions. The definitions of `FloatLiteral` and `Expr` have been omitted for brevity; in essence they match floating point numbers such as 3.141, simple expressions like `2.0 PI` (2π), and also arbitrary identifiers like `x`, `A`, `var_name`.

```
Model ExampleLocal
Frames {
 Sxyz, Exyz,  // X - Y - Z
 Sxzy, Exzy,  // X - Z - Y
 Syxz, Eyxz,  // Y - X - Z
 Syzx, Eyzx,  // Y - Z - X
 Szxy, Ezxy,  // Z - X - Y
 Szyx, Ezyx,  // Z - Y - X
 A, B
}
Convention = local
// Euler angles, intrinsic rotations
Sxyz -> Exyz : rotx(rx) roty(ry) rotz(rz)
Sxzy -> Exzy : rotx(rx) rotz(rz) roty(ry)
Syxz -> Eyxz : roty(ry) rotx(rx) rotz(rz)
Syzx -> Eyzx : roty(ry) rotz(rz) rotx(rx)
Szxy -> Ezxy : rotz(rz) rotx(rx) roty(ry)
Szyx -> Ezyx : rotz(rz) roty(ry) rotx(rx)
// Some translations
A -> B : trx(tx) roty(1.37) rotz(rz) trz(0.55)
```

Figure 6.5 – A possible document compliant with the motion DSL. This example lists all the six possible combinations of three successive rotations of a rigid body. The last line shows a motion that also includes translations.

```
Model:
    "Model" name=ID
    framesList = FramesList
    "TransformedFramePos""="convention=("left"|"right")
    transforms += Transform*
;

FramesList:
    "Frames""{"
        items += Frame
        (','items += Frame)*
     "}"
;

Frame: name=ID;

Transform:
    '{'leftFrame=[Frame]'}''_X_''{'rightFrame=[Frame]'}'
    '=' matrices += AbsMatrix*
    ('['userName=ID']')?
;
```

```
AbsMatrix:
    Rotation | Translation
;
Rotation:    Rx | Ry | Rz ;
Translation: Tx | Ty | Tz ;

Tx:  "Tx""("arg=ArgSpec")";
Ty:  "Ty""("arg=ArgSpec")";
Tz:  "Tz""("arg=ArgSpec")";

Rx:  "Rx""("arg=ArgSpec")";
Ry:  "Ry""("arg=ArgSpec")";
Rz:  "Rz""("arg=ArgSpec")";
```

Figure 6.6 – The grammar of the DSL for coordinate transforms. See Figure 6.4 for the description of the `ArgSpec` element.

```
Model ExampleLocal_left
Frames {
    Sxyz, Exyz, Sxzy, Exzy, Syxz, Eyxz,
    Syzx, Eyzx, Szxy, Ezxy, Szyx, Ezyx, A, B
}
TransformedFramePos = left

{Sxyz}_X_{Exyz} = Rx(-rx) Ry(-ry) Rz(-rz)
{Sxzy}_X_{Exzy} = Rx(-rx) Rz(-rz) Ry(-ry)
{Syxz}_X_{Eyxz} = Ry(-ry) Rx(-rx) Rz(-rz)
{Syzx}_X_{Eyzx} = Ry(-ry) Rz(-rz) Rx(-rx)
{Szxy}_X_{Ezxy} = Rz(-rz) Rx(-rx) Ry(-ry)
{Szyx}_X_{Ezyx} = Rz(-rz) Ry(-ry) Rx(-rx)
{A}_X_{B} = Tx(-tx) Ry(-1.37) Rz(-rz) Tz(-0.55)
```

(a) Convention left

```
Model ExampleLocal_right
Frames {
    Sxyz, Exyz, Sxzy, Exzy, Syxz, Eyxz,
    Syzx, Eyzx, Szxy, Ezxy, Szyx, Ezyx, A, B
}
TransformedFramePos = right

{Sxyz}_X_{Exyz} = Rx(rx) Ry(ry) Rz(rz)
{Sxzy}_X_{Exzy} = Rx(rx) Rz(rz) Ry(ry)
{Syxz}_X_{Eyxz} = Ry(ry) Rx(rx) Rz(rz)
{Syzx}_X_{Eyzx} = Ry(ry) Rz(rz) Rx(rx)
{Szxy}_X_{Ezxy} = Rz(rz) Rx(rx) Ry(ry)
{Szyx}_X_{Ezyx} = Rz(rz) Ry(ry) Rx(rx)
{A}_X_{B} = Tx(tx) Ry(1.37) Rz(rz) Tz(0.55)
```

(b) Convention right

Figure 6.7 – Two documents compliant with the transforms DSL, both showing the coordinate transforms corresponding to the rigid motions shown in Figure 6.5. Note the sign inversion in the arguments of the primitive transforms, because the two documents use a different convention but model the same information (see Section 5.2.2.1).

propagated in the generated code, where it will eventually have to be resolved to an actual value (with any mechanism that is not part of the DSL infrastructure).

The value of the `TransformedFramePos` property of the document (that is either `left` or `right`) basically sets the `convention` property (see Section 5.2.2) of all the atomic transforms used in the document (as in the previous language, the property is global to the document rather than being an attribute of each transform, in order to simplify the notation). This property is fundamental to enable the generation of correct code implementing the matrices; the same sequence of simple transforms translates in general to different code, if the convention changes.

6.2 Code generation

This section finally describes the code generators implemented on top of the languages described before. With little additional logic to orchestrate the code generators of each language, we obtain a rich generator specific for articulated robots, which can produce various implementations of dynamics and kinematics algorithms. This point is the subject of the last sub-section (6.2.3), while the preceding ones describe the single generators.

A technical note To understand better the whole functioning of the code generation framework based on Xtext, it is worth explaining that the grammar of the languages not only defines the content of compliant text documents, but also serves as a plan for the parser of the language to create the run–time data structure matching the parsed document. In other words, given a valid input file, say one of the kinematics DSL, the language infrastructure will create a data structure that matches such a description, with first–class objects representing the joints, links, etc. This structure will then be available to the code generator software, which can effectively take it as an input during the generation process.

6.2.1 Robot–specific dynamics routines

One of the crucial points about the code generation for dynamics algorithms is that the generator is provided with a full model of a robot, with which it can resolve the dependency of the algorithms on such a model (see Section 5.1.2.3). Resolving the dependency means that the generated implementation is no longer parametrized with the robot description, but that information is implicitly embedded into it (see Figure 6.8).

Robot–specific implementations have the advantage of being faster since they realize a simplified version of the algorithm, stripped of all the logic required to be general. For example there is no need to navigate at run–time a data structure describing the robot geometry, making loops and boolean tests unnecessary. Prior knowledge about the structure and the parameters of the robot also allows to do numerical optimizations, like avoiding useless operations. For instance, in the assumption of having only plain prismatic or revolute joints, the matrix $\mathbf{S}$ describing the motion subspace of a joint is a single column vector with only one non-zero element, thus operations with this matrix can be greatly simplified (Featherstone 2008).

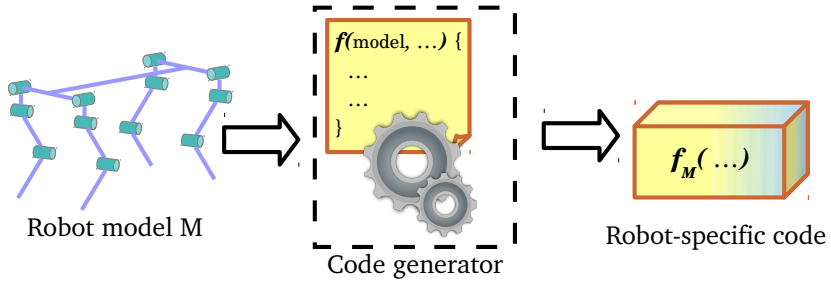


Figure 6.8 – The generation of the implementation of the dynamics algorithms tailored for specific robots. The code generator must embed the knowledge about the abstract algorithm. A concrete implementation is achieved by turning the algorithm into code in some language, where the references to the robot model are substituted with actual values and specific instructions.

Another meaningful example of the benefits of this approach is related to the *branch–induced sparsity*; this term refers to the sparsity pattern (the presence of constant zeros in fixed positions) observable in certain matrices involved in kinematics/dynamics computations, as a consequence of the branches in the kinematics structure of the robot (Featherstone 2008, 2010c). The performance of computations can benefit from the sparsity both during the computation of the matrix itself, as well as in the operations involving the same matrix.
Specific algorithms – such as those described in the works just cited – take advantage of how the branches induce the sparsity, so they skip the computations for the elements known to be zero; therefore they are usually more efficient than the general purpose ones. Note that this is also an example of the need to look into the robot structure (in this case to identify branches), something that with our approach happens off–line, in the code generator itself, as we said before. Therefore the generated code can exploit both things combined: sparsity patterns and prior knowledge about the robot structure. It does not contain *any* instruction for the zero-elements, nor it has to find at run–time which are such elements.

Currently our system can generate efficient code for the *Composite–Rigid–Body algorithm* for the calculus of the Joint–Space Inertia Matrix (JSIM) $\mathbf{H}$, which indeed exhibits a sparsity pattern due to kinematic branches (see also Section 5.1.2.2). It can also generate efficient code that computes the inverse $\mathbf{H}^{-1}$; this code implements the $\mathbf{L}^T\mathbf{L}$ factorization of $\mathbf{H}$, where $\mathbf{L}$ is a lower triangular matrix that preserves the same sparsity pattern as $\mathbf{H}$ (Featherstone 2005), and a custom routine for the inversion of a lower triangular matrix that takes advantage of the sparsity as well.
In general, the dynamics algorithms we support are known in the literature to be the fastest for their task, and the generated code includes the minimum set of instructions implementing them for a specific robot; therefore, the generated code has in theory a run–time performance that is close to the maximum achievable. The high performance of these implementations and the purely numerical nature of the algorithms (e.g. no I/O operations, unless naively implemented) allow to use some of the generated code (the C++ code in our case) in hard real–time controllers running at high frequencies.

Note that the approach of code generation for robot–specific implementations basically solves the apparent tradeoff between generality/flexibility and efficiency: usually a general purpose library or similar (like a dynamics engine) can be used for a variety of inputs but is not as efficient as a dedicated implementation. On the other hand developing tailored procedures for each specific case sacrifices re–usability and increases the software development effort. Trading off these two qualities is indeed a known issue in software engineering (Ghezzi et al. 2002).
However, a code generation process does not have such drawbacks and solves the trade off, since it is still flexible enough to be used for a wide class of inputs (it can be used with any articulated robot) yet provides the benefits of optimized implementations.

6.2.2 Coordinate transforms

As explained before in the text, coordinate transforms are a fundamental part of almost any software for articulated robots, but at the same time the calculus and the coding of such objects can be very tedious and error prone. Moreover, the need for efficient implementations described in Section 4.2.1 still holds, since they are also used in real–time controllers and anyway faster code is always preferable, if it does not come with an additional cost. Therefore coordinate transforms are also an ideal target for code generation, and our framework supports this feature.

6.2.2.1 Rationale

The idea behind the motion DSL and the transforms DSL and their generators is that the user specifies some high level input information, for example a sequence of roto–translations a body has undergone, and she receives as output the code implementing some coordinate transform matrices of interest. For example she specifies the pose of a reference frame $\mathfrak{F}_A$ fixed on a rigid body with respect to frame $\mathfrak{F}_O$ fixed on the same body, and she gets the MATLAB code for the 6×6 spatial–velocity transform from $\mathfrak{F}_O$ to $\mathfrak{F}_A$.[2] The user deals only with high level information and he does not have to perform tedious calculus and manual implementation that are usually of no interest, but still require a lot of effort.
Basically the user has to be relieved from doing two macro–tasks:

1. Find the appropriate sequence of simple coordinate transforms that match the relative pose between the two frames.

2. Find the actual value of each of the simple transforms, develop the multiplications and get the resulting matrix; implement this with a programming language, possibly making it parametric with respect to possible changes in the pose.

As explained in Section 5.2.2 both these tasks can be tricky and confusing, when performed manually.

[2] This is the matrix that multiplies the coordinate vector with the linear and angular velocity *of* frame $\mathfrak{F}_O$, giving as a result the velocity *of* frame $\mathfrak{F}_A$. The two velocity vectors express the *same* motion of the body (which is obviously unique, at each instant).

While in principle it could be possible to automate all these tasks within the infrastructure of the motion DSL (since all the information is in its documents), two distinct languages allow to partition the implementation of the code generator, identifying two distinct roles:

- The code generator of the motion DSL deals with the first task, that is it transforms a user written description of a rigid motion into the correct sequence of coordinate transforms. This relieves the user from remembering how the matrices are composed, which also depends on the convention the primitive motions are expressed with (see again Section 5.2.2).
  Since the other language is designed exactly to write sequences of coordinate transforms, the code generator of the motion DSL can very well generate just a document that is compliant with the transforms DSL.

- The code generator of the transforms DSL, on the other hand, creates source code (potentially in any language) with the correct numerical implementation of the coordinate transforms listed in its documents. This is possible since any simple transform is uniquely determined, when the value of the convention attribute is set (see Section 5.2.2.1).

Note that the transforms DSL alone is also useful if for some reason the user wants to explicitly input a sequence of transformation matrices instead of a rigid motion, that is the user is only concerned with the second of the tasks listed above; this might happen for example for tests or comparisons with existing implementations (e.g. to compare a manually developed matrix with the generated one).
In general the distinction between the two languages and the corresponding code generators simply reflects the fact that rigid motions and coordinate transforms are two distinct problems, though tightly correlated.

6.2.2.2 Basic strategy

The basic approach to generate code for the transforms simply consist in translating the atomic transforms into matrices in the target language, and the composite transforms into the actual products of such matrices. This approach is very convenient when a small library in the target language that already implements the six basic transforms `Rx()`, ..., `Tz()` is available: in this case the job basically reduces to generate a sequence of products of the identifiers corresponding to the atomic transforms. This is the technique that we use to generate code for the symbolic engine – see below.
In this case the software needs to know (e.g. by means of a configuration file) the convention used by the library to be referenced in the generated code, and compare it with the one adopted in the input document. In this way it is able to decide whether the abstract element $Rz(\alpha)$, for instance, correspond to its counterpart in the library or its inverse. This is a practical demonstration of the need of a proper identification of the relevant properties of the objects of interest, and of the need to explicitly expose such properties, an issue discussed in the section about the domain model (5.2.2.1).
A more sophisticated approach for the code generation, which is the one actually implemented in our framework, is described below in Section 6.2.2.3.

6.2.2.3 The role of the symbolic computation engine

A symbolic engine is a software capable of manipulating mathematical expressions as such, applying the rules of algebra and of more complex calculus. For example, it can develop a matrix–vector product where the elements of the two operands are simply symbols or functions, like x, a, $cos(2\theta)$, without assigning a numerical value to them.

In our framework the symbolic engine is used to achieve higher efficiency in the code generated for coordinate transforms. The idea is to use the engine to develop the multiplications between the atomic transforms and get the resulting matrix, possibly simplified to a compact form, as an explicit step of the code generation process. Therefore the engine must be able to perform linear algebra and possibly some simplifications of trigonometric functions.
Then, a generator for another language can take advantage of prior knowledge about the structure of the resulting matrix: which elements are constant, which ones are equal to some other, which and how many unique trigonometric functions the matrix depends on, so that the generated code does not compute the same thing multiple times (like sines and cosines, typically the most expensive functions) and does not compute at all the constant elements.

Code for the symbolic engine itself is generated with the standard approach described above, to have the engine develop the products in a second step. Other languages are addressed during subsequent steps.

Some technical notes The symbolic engine we have employed in our workflow is called Maxima, it is open source, based on the LISP language, and can be built for a variety of platforms (Maxima 2011). It is maybe not as sophisticated as some commercial alternatives, but it is more than enough for our needs, and it satisfies our project constraint of implementing the whole framework with open source technologies only.
A main drawback of Maxima– that would not necessarily be solved by other commercial engines though – is that it is primarily meant to be used as a command line tool interacting with the user; textual results are obtained in response to textual input commands. In order to use the engine programmatically within the code generator software, we adopted the Jacomax wrapper, which provides a Java interface to access the facilities of the engine and runs it in a separate process (McKain 2013).
However, this does not solve the issue of text–based input/output, so that commands and especially results have to be plain strings, making the pieces of the code generator interacting with Maxima a bit cumbersome and awkward.

To achieve a robust and general method to deal with the textual output of Maxima, instead of digging into the internals of the engine and the LISP language, we developed yet another simple Xtext grammar specifically tailored for this purpose: output expressions of Maxima (which in our case are limited to matrices whose elements are trigonometric functions) happen to be compliant with the language, so that it provides us with a proper parser that constructs a first–class object representation of the textual expressions (i.e. a strongly typed data structure that mirrors the analytical expressions, telling which sines and cosines are there, how they are composed, which are the arguments). We can therefore analyze the expressions, find which optimizations are possible and transform them into other languages in a principled way. Figure 6.9 shows a

short example of a document of this language.

```
Variables {
    haa, hfe, kfe
}

- 0.165 * cos(- kfe + hfe + haa) - 0.165 * cos(kfe + hfe - haa);
- 0.165 * cos(kfe + hfe + haa) + 0.165 * cos(kfe + hfe - haa);
cos(kfe + hfe + haa);
 - sin(-haa);
- 1.234E-6 * cos(kfe);
```

Figure 6.9 – An example of a text document compliant with the DSL designed to parse MAXIMA expressions. It contains the declaration of the referenced variables and then a list of arbitrary long algebraic expressions that can include sines and cosines.

6.2.2.4 A complete example

Figure 6.10 illustrates most of the steps involved in the actual generation of executable implementation of coordinate transformation matrices:

1. The starting point may be a document of the motion DSL specifying a set of rigid motions that relate the pose of various reference frames – see Figure 6.10a. When working with a robot model, this file itself can be automatically generated and it contains the geometrical parameters of the robot (see Section 6.2.3 and Figure 6.13).

2. From the information about rigid motions, an abstract specification of coordinate transforms can be generated, in the form of a transforms DSL document. The transforms to appear in this document can be chosen with a configuration file (see Figure 6.11). Note that it is possible to get the transform between frames that are "connected" by some rigid motion, even if such motion does not appear explicitly in the motion DSL document; for example, two motions like `A->B` and `B->C` allow to ask for the transform between `A` and `C`. See Figure 6.10b.

3. The code generator software of the transforms DSL can turn the specification of transforms into MAXIMA code, specifically into the proper sequence of products between the MAXIMA matrices implementing the basic transforms (see Section 6.2.2.3). Figure 6.10c shows how MAXIMA displays the resulting matrix after developing the products and doing some simplifications (plus some variable substitution for the sake of displaying the code in the figure).

4. Finally, after generating the code for the symbolic engine, the engine itself is invoked by the generator for another language, to access the resulting matrix and turn it into code for that language. The example in Figure 6.10d shows MATLAB code, and specifically the set of assignments that update the matrix (called `X`). The sines and the cosines of the status of each revolute joint have to be computed as well; the corresponding code is generated but it is not shown for brevity.

```
Model HyL_frames
Frames {
 fr_base,fr_slider,fr_hip,fr_leg,fr_lowerleg,
 fr_foot, fr_SLIDE,fr_HAA,fr_HFE,fr_KFE
}
Convention = local

fr_base   -> fr_SLIDE :
fr_slider -> fr_HAA : roty(-PI/2.0) rotz(-PI)
fr_hip    -> fr_HFE : trx(0.08) rotx(PI/2.0)
fr_leg    -> fr_KFE : trx(0.35)

fr_SLIDE -> fr_slider   : trz(q_SLIDE)
fr_HAA   -> fr_hip      : rotz(q_HAA)
fr_HFE   -> fr_leg      : rotz(q_HFE)
fr_KFE   -> fr_lowerleg : rotz(q_KFE)

fr_lowerleg -> fr_foot : trx(0.33)
```

(a) The positions of the various frames of the robot, in a document of the motion DSL.

```
Model HyL_transforms
Frames {
    fr_base, fr_foot //, ...
}
TransformedFramePos = right

{fr_base}_X_{fr_foot} =
        Tz(q_jSLIDE) Ry(-PI/2.0) Rz(-PI) Rz(q_jHAA)
          Tx(0.08) Rx(PI/2.0) Rz(q_jHFE)
            Tx(0.35) Rz(q_jKFE) Tx(0.33)
// ...
```

(b) The specification of a transform, generated from (a), in a document of the transforms DSL.

```
[     -c_HFE s_KFE -c_KFE s_HFE             s_HFE s_KFE -c_HFE c_KFE            0
[
[s_HAA s_HFE s_KFE -c_HFE c_KFE s_HAA  c_HFE s_HAA s_KFE +c_KFE s_HAA s_HFE   c_HAA
[
[c_HAA s_HFE s_KFE -c_HAA c_HFE c_KFE  c_HAA c_HFE s_KFE +c_HAA c_KFE s_HFE  -s_HAA
[
[                  0                                       0                      0

                        -0.33 c_HFE s_KFE -0.33 c_KFE s_HFE -0.35 s_HFE           ]
                                                                                  ]
      0.33 s_HAA s_HFE s_KFE -0.33 c_HFE c_KFE s_HAA -0.35 c_HFE s_HAA -0.08 s_HAA]
                                                                                  ]
0.33 c_HAA s_HFE s_KFE -0.33 c_HAA c_HFE c_KFE -0.35 c_HAA c_HFE -0.08 c_HAA + sld]
                                                                                  ]
                                            1.0                                   ]
```

(c) The matrix as computed by the symbolic engine MAXIMA, by multiplying together all the factors visible in Figure (b); `sld` is the status of the slider. The fourth column is displayed below the first three to fit the page.

```
X(1,1) = (- c_HFE * s_KFE) - (s_HFE * c_KFE);
X(1,2) = (s_HFE * s_KFE) - (c_HFE * c_KFE);
X(1,4) = (- 0.33 * c_HFE * s_KFE) - (0.33 * s_HFE * c_KFE) - (0.35 * s_HFE);
X(2,1) = (s_HAA * s_HFE * s_KFE) - (s_HAA * c_HFE * c_KFE);
X(2,2) = (s_HAA * c_HFE * s_KFE) + (s_HAA * s_HFE * c_KFE);
X(2,3) = c_HAA;
X(2,4) = (0.33 * s_HAA * s_HFE * s_KFE) - (0.33 * s_HAA * c_HFE * c_KFE)
          - (0.35 * s_HAA * c_HFE) - (0.08 * s_HAA);
X(3,1) = (c_HAA * s_HFE * s_KFE) - (c_HAA * c_HFE * c_KFE);
X(3,2) = (c_HAA * c_HFE * s_KFE) + (c_HAA * s_HFE * c_KFE);
X(3,3) = - s_HAA;
X(3,4) = q(1) + (0.33 * c_HAA * s_HFE * s_KFE) - (0.33 * c_HAA * c_HFE * c_KFE)
          - (0.35 * c_HAA * c_HFE) - (0.08 * c_HAA);
```

(d) The Matlab code to update the transform matrix `X`. The terms {`c|s`}`_XXX` must be updated every time the joint status vector **q** changes (`q(1)` is the status of the slider). No instruction is generated here for the constant elements of the matrix; those assignments are contained in another file to be run only once.

Figure 6.10 – The steps involved for the generation of code for coordinate transformation matrices. The examples refers to the transform from the foot to the base of the HyL robot, a hydraulic leg of HyQ (see Figure 6.3 for the meaning of the labels). HyL is a fixed–base robot with a prismatic joint allowing vertical movement (a slider). The terms {`c|s`}`_XXX` represent the cosine and the sine of the joint status variable `XXX`.

Figure 6.11 shows an example of a document of another very simple language we designed to write configuration files, with which the user can specify the desired transformation matrices whose implementation has to be generated.

```
Robot Planar
Frames {
    fr_Base, fr_link1, fr_link2, fr_ee
}

Transforms {
    base=fr_link1, target=fr_ee
    base=fr_link2, target=fr_ee
}

Jacobians {
    base=fr_Base, target=fr_ee
}
```

Figure 6.11 – A simple configuration file to declare the coordinate transforms and the Jacobians of interest for the user. They are identified by the name of the two reference frames they refer to. Note that requesting a particular Jacobian might trigger the generation of additional transforms that are not explicitly listed in this file, since the computation of the Jacobian depends on them. This example has been used to get the Jacobian matrix shown in Figure 5.2

6.2.3 Putting all together: the robotics code generator

Figure 6.12 gives an overview of all the components and the data flows involved in the generation of robot–specific code, which is in fact a particular usage of the infrastructure provided by our specification languages.
As expected, the main information source at the foundation of the process is the kinematics and dynamics model of an articulated robot. The code generator of the kinematics DSL is directly implementing the generation of the dynamics algorithms, as illustrated above, but is also orchestrating the other generators for the coordinate transforms, as if it was a user of the other languages. Note that the dynamics algorithms themselves depend heavily on the coordinate transforms generated separately, which is yet another reason for striving for efficiency also for the transforms.

As illustrated in the figure, another direct product of the generator of the kinematics DSL is the specification of the geometry of the robot, extracted from the position parameters of the joint frames, in the form of a document of the motion DSL; that is the kinematics DSL generator writes an instance document of the motion DSL. As a matter of fact, the six parameters about the pose of the joint frames in the kinematics DSL document are nothing else than a specification of a rigid motion, which can be equivalently expressed with the motion DSL. The generation of the motion DSL document is really just a change of representation of the same information already encoded in the kinematics DSL document (see Figure 6.13). In the kinematics DSL we use a custom representation (the six parameters as explained in Section 6.1.3) to make the document more readable and keep the grammar simple.

This approach has the advantage that the code generators of the motion DSL and the transforms DSL can be completely reused, and no additional logic

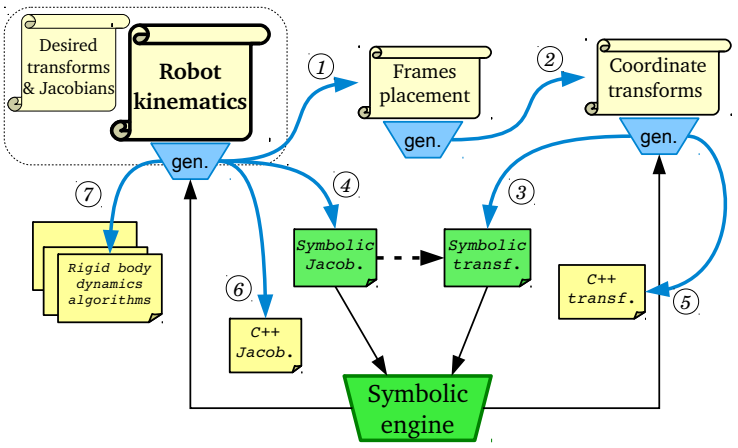


Figure 6.12 – Overview of the robotics code generator workflow. The scroll-like blocks are instance documents of the DSLs (left to right: the configuration DSL, kinematics DSL, motion DSL, transforms DSL); the trapezoidal blocks perform computation, while the sheet-like ones are resulting code blocks. The curved blue arrows illustrate a code generation step, while the straight black ones are generic input/output data flows (the dashed arrow shows the dependency of the symbolic Jacobians on the transforms). The numbers give the reader an idea about the sequence of the steps, even though certain things do not have dependencies, like dynamics algorithms.

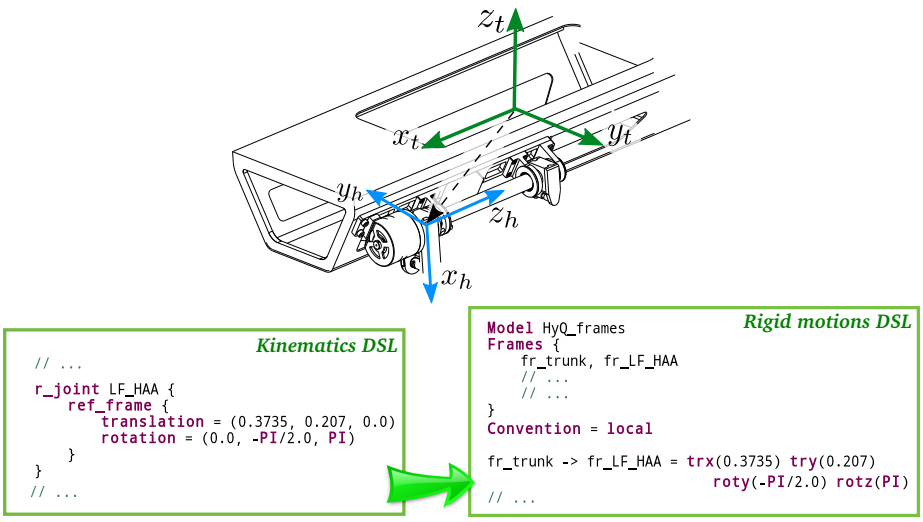


Figure 6.13 – The transformation of the joint positioning parameters in the kinematics description to a document of the motion DSL. This step is purely a change of the representation of the same information. This example refers to the `HAA` (Hip Abduction Adduction) joint for the left-front leg of the robot HyQ; the frame of the joint (in blue) can be reached from the frame of the trunk (in green) with a translation of about $(0.37, 0.2, 0)$ meters, followed by a rotation of $-\pi/2$ about the y axis and π about the z axis.

has to be added (actually replicated) in the kinematics DSL infrastructure. The generation of the implementation of the coordinate transforms then happens as explained in Section 6.2.2: the motion DSL document containing the geometry information serves as the main input to generate a list of abstract transforms in the form of a transforms DSL document, which also depends on the configuration file the user fills with the list of the desired transforms (e.g. the transform from the foot frame to the trunk frame, in a humanoid robot).[3]
If the desired transforms refer to frames belonging to separate links on a kinematic chain – which is very likely to happen – then they will be defined with a dependency on some variables that correspond to the status of the joints on the same kinematic chain.

6.2.3.1 Geometric Jacobians

The code generation for geometric Jacobians has been implemented directly in the kinematics DSL infrastructure, since no dedicated language was necessary for the purpose. And if calculus related to rigid motions and coordinate transforms does not necessarily require a robot model, on the other hand Jacobians are always defined for points of a multibody system, such as a robot, so it is sensible to support them within the kinematics DSL.
As described in Section 5.2.2.2, Jacobians are uniquely identified by a pair of reference frames as for the coordinate transforms, and those desired by the user can be specified in the same configuration file used for transforms (see Figure 6.11), with a very similar syntax; the section `Jacobians` has to be added at the bottom of the file.

The actual code generation happens in a way quite similar to coordinate transforms, since Jacobians are also matrices and we can use the symbolic engine to generate optimized code. For each Jacobian the software generates code for the symbolic engine that in turn depends on the symbolic code for the coordinate transforms. This code basically implements the algorithm to compute a geometric Jacobian from the forward kinematics functions of the robot, which is nothing else than coordinate transforms. For example, the computation of a Jacobian with respect to frame $\mathfrak{F}_B$ requires the direction in space of all the joint axes in the path from $\mathfrak{F}_B$ to the point of interest: such information can be extracted by the third column of the homogeneous transforms (i.e. the direction cosine for the z axis, which is always lying on the joint axis by convention) from the frame of each joint to $\mathfrak{F}_B$.
During a second step the engine is required to interpret this code so that an actual matrix is obtained, and this matrix can then be translated into code with the same mechanisms seen before for the coordinate transforms.

[3] Some coordinates transforms are always generated regardless of the user configuration, since they are required by the dynamics algorithms.

Chapter 7

Experimental results

7.1 Control software for articulated robots

This section gives an overview of the main results achieved on the HyQ robot also thanks to the implementation of all the software components described in chapter 4. The purpose of the section is to demonstrate that the proposed design works, is effective, and compatible with the strict real–time requirements of the motor control of an articulated robot. It has been used to bring a real, sophisticated robot to the stage in which it can be controlled by computer code and can be used to address open research questions, e.g. concerning artificial legged locomotion.

Figure 7.1 shows several snapshots of the HyQ robot when performing some walking experiments. In this case – as well as in any other motion – some higher level logic computes the trajectories to be followed by the legs as a sequence of set–points for the positions of the joints, which are transformed into commands to be sent to the hydraulic valves; this mapping from desired positions to actuator commands is done through a control process that continuously monitors the actual position of the joints, to determine more accurately how much the valves have to be opened in one of the possible directions. Therefore the I/O with the hardware is a critical section of the control process.
Besides showing that the robot can actually perform non–trivial motions, this example is meaningful in reference to the flexibility of the I/O library. The pictures in Figure 7.1 show two different walking experiments performed while the robot is equipped with two completely different types of hydraulic valves (with respect to internal functioning, input commands, performance, etc.). After the hardware update (i.e. the change of the valves) the whole client code working on top of the hardware I/O does not need any particular change since the port of interaction with the hardware was designed since the beginning as a generic interface (Figure 7.2 shows the simple interface for hydraulic valves). The only necessary changes are confined in the concrete implementation of the interface, which has to deal with the details of the specific hardware model. A new tuning of the gains of the controllers is also likely to be required since the system as a whole and so its dynamic behavior have changed, but this point is related to the functioning of the controllers and is basically unavoidable (unless more sophisticated, adaptive or learning controllers are used).

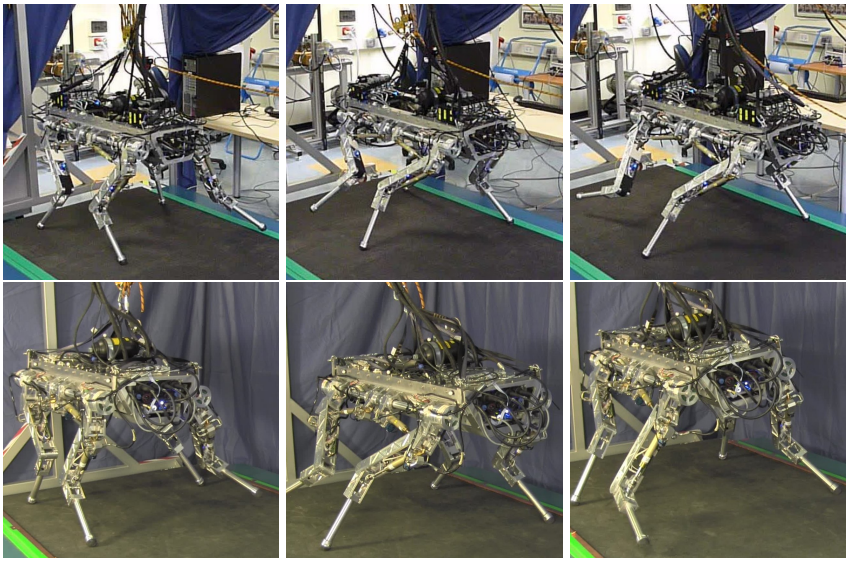


Figure 7.1 – Snap shots of walking experiments performed with the HyQ robot, before and after the replacement of the hydraulic valves. Thanks to the modular design of the hardware I/O library, this change did not affect the client code responsible for the generation of the feet trajectories, which could be entirely reused.

```
class Valve {
public:
    virtual ~Valve() = 0;

    /**
     * Identifiers for the two ports of a typical hydraulic valve.
     */
    enum Port{ A=0, B=1, _PORTS_COUNT };

    /**
     * Opens one of the ports of the valve
     * \param amplitude the amount of the opening, between 0 and 1
     * \param port the port to be opened
     */
    virtual double go(double amplitude, Port port) = 0;

    /**
     *  Forces the closure of this valve.
     */
    virtual void stop();
};
```

Figure 7.2 – The simple C++ interface designed to abstract the hydraulic valves used to actuate the legs of the HyQ robot. `A` and `B` are the names commonly used in the literature to identify the two ports of a hydraulic valve, i.e. the two possible directions in which the oil can flow and so the directions of motion of the actuator attached to the valve. Somewhere else in the software system there must be some sort of configuration mapping the direction of motion of the joints with the ports of the corresponding valves, matching the conventions and the physical connections on the robot.

As described in Chapter 4, the first layer addressing the hardware abstraction for the HyQ robot has been designed to be general and robot–independent, so that it can be used in different robots that share some of the sensors and/or actuators. We have indeed installed it on some platforms used to support the research on HyQ, such as an exact copy of the hydraulic leg constrained to a vertical slider.
This robot is mainly used to investigate the behavior of hydraulic actuation and the corresponding controllers, and it uses the same encoders, force sensors and hydraulic valves mounted on the quadruped, as well as an analogous computer and the same technology for the low level I/O of analogue and digital signals. Figure 7.3 shows some snapshots taken during a hopping experiment with the

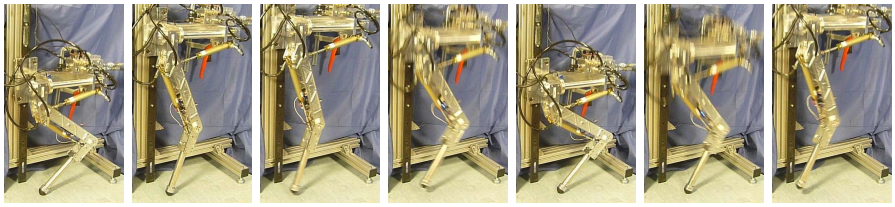
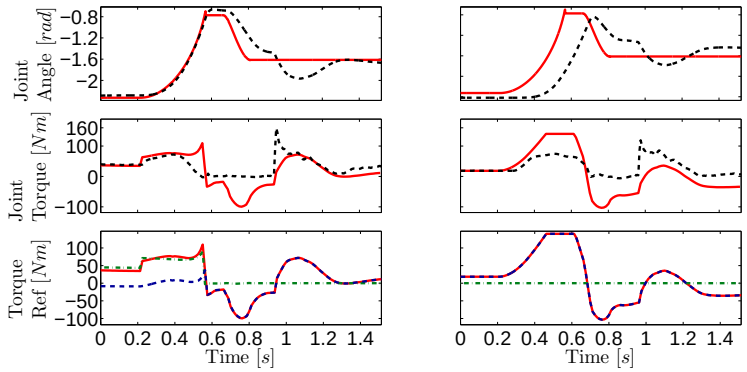


Figure 7.3 – Snap shots of a hopping experiment performed on the hydraulic leg of the HyQ robot. The computing system controlling the robot uses the same hardware I/O library described in Section 4.2.2, designed primarily for the HyQ but re–usable on analogous hardware. Experiments on the single hydraulic leg allowed to gain deeper insights about the behavior of hydraulic actuation, and develop effective hydraulic controllers (Boaventura, Semini, Buchli, Frigerio, et al. 2012).

leg, that was possible thanks to the same hardware I/O library implemented for HyQ (Boaventura, Semini, Buchli, and Caldwell 2011). The low level control loop used for this experiment, whose process also has to sample position/force sensors and send commands to the valves, is running with a frequency of 1 KHz with hard real–time requirements, therefore the I/O components of the library have to be compatible with such strict requirements. Obviously, only the robot–specific part of the hardware abstraction had to be rewritten, to match the number of sensors, the actual electrical connections with the I/O boards, etc. Basically this part can be seen as a mere configuration layer.
Similarly, some components of the I/O library were re–used in other simpler platforms also designed to investigate the hydraulic technology; specifically, the software wrappers for the hydraulic valves, as part of the library, have been used on a mechanism with a single hydraulic cylinder moving a small cart constrained on a rack, and for another simple mechanism mounting a hydraulic rotary actuator, which is currently undergoing some tests.

Figure 7.4 shows the position and the force tracking at the right hind knee joint of the HyQ robot, during a squat jump (Boaventura, Semini, Buchli, Frigerio, et al. 2012; Semini, Khan, et al. 2012). The purpose of this figure is to show the capability of our software system to control the robot during field experiments. In addition, since the plots were originally meant to show the accuracy of the force control and the dynamic model of the robot, they also show the importance of inverse dynamics for fast and dynamic movements such as a jump.



(a) Inverse dynamics active.

(b) Inverse dynamics never active.

Figure 7.4 – Position and force tracking of the right hind knee joint during a squat jump experiment with the HyQ robot. The initial acceleration phase lasts until $0.55s$; during this interval, the inverse dynamics was activated. In the first two plots, the red solid lines indicate the reference command (——), while the black dashed ones the actual value (**- -**). The last plot shows how the reference torque τ_{ref} is generated as the sum of the feed–forward (**- -**) and the position error feedback (**- -**) term: $\tau_{ref} = \tau_{ff} + \tau_{fb}$. The reference torque is almost completely obtained by the feed–forward term computed by the inverse dynamics algorithm.

Therefore Figure 7.4 provides an evidence about the importance of model based control for high performance behaviors in articulated robots, which is one of the motivations behind the development of our code generation framework.

The last picture sequence, in Figure 7.5, shows a few snapshots of the robot performing more advanced tasks, such as trotting, traversing moderately rough terrain and responding to lateral disturbances such as kicks. The purpose of

Figure 7.5 – The HyQ robot performing a few more advanced tasks. From left to right: reacting to sideways disturbances, trotting over small obstacles on a treadmill, trotting outside on rough terrain.

these images is once again to demonstrate the effectiveness of the control software system, also for more sophisticated robot behaviors that require a non trivial logic. All the details about the methods implemented to achieve such

results can be found in (Barasuol et al. 2013).
The robot behavior logic underlying the actions illustrated in the figure is actually pushing the limits of the architecture described in Chapter 4, because of the complexity and the variety of the algorithms, making a more detailed design of the internal structure of the robot behavior block necessary. The most evident limitations of the current system will be described in the conclusions, Section 8.1.
However, as detailed in the reference mentioned before, the control algorithms running during the experiments of the figure are heavily based on kinematics and dynamics computations, confirming once again their importance in the implementation of sophisticated behaviors of articulated robots. This is indeed one of the motivations behind the development of our code generator framework, whose preliminary results are shown in the next section.

7.2 Evaluation of the code generator

This section discusses the concrete usage of the DSLs infrastructure described in Chapter 6 and the benefits for the end user. Most of the numerical results concern primarily the performance of the generated code, since it is difficult to find metrics and thus tests to measure usability and flexibility, besides an explicit description and some examples about these features. For these tests we usually compare C++ implementations, but our generator can currently emit also MATLAB and MAXIMA code; at the price of developing more generator templates, potentially any language can be supported.

7.2.1 General remarks

As far as the usage of the languages is concerned, results from our experience are promising. With the kinematics DSL, for example, creating new robot descriptions is a matter of minutes, since the DSL is simple and intuitive; most of the time is typically spent looking in the robot documentation for the inertia parameters and the geometrical parameters. Once the code generators are properly verified, by means of trials, tests and comparisons, it is impossible to introduce bugs such as memory leaks or logical errors in the implementations for kinematics and dynamics.
Another benefit of our approach is that the code generator can give guarantees about the consistency between its diverse outputs (i.e. source code in different languages), which have been produced given the same input information (e.g. a robot model). It is then possible, for instance, to generate exactly the same Jacobian matrices in C++ and in MATLAB, using the second to quickly verify them and try them in rapidly prototyped algorithms such as controllers. Afterwards the C++ Jacobians can be used in the software deployed on the embedded computer controlling a real robot, knowing that they will behave exactly as the MATLAB counterparts.
These guarantees are of great significance to the user who can be relieved from the manual analysis of the generated code and can then focus the debugging on the models, that are simpler hence easier to inspect, or on other parts of the system.

However, the trust of the user for a new tool requires time, during which the

correctness of the tool itself is not given for granted. To support the end user in this phase, but also the same developers of the code generation framework, it is valuable for the whole toolchain to be as clear and as observable as possible. To a certain extent, this feature is achieved by having simple languages that target well defined and confined problems, so that it is relatively easy to perform many trials and understand the behavior of the generators. The clarity in the generated code is also of great help in the debugging and also to give the user more chances to understand and thus be satisfied with the tool. For example, the generated code in C++ – which is what we focus on in these paragraphs, since it is the fastest code and it can also be used in real–time controllers – uses the Eigen linear algebra library: Eigen is a modern, carefully designed and quite well documented library for efficient computations with matrices, which allows to have much more compact and readable code for algebra operations rather than custom solutions, without thereby compromising efficiency (Guennebaud et al. 2013). Quite the opposite, Eigen is specifically designed with performance in mind, and with proper use of its facilities code for hard real–time controllers can be developed.

7.2.2 Validation

In order to validate the numerical correctness of the generated code we have tested them against other existing and established implementations, automating the comparison of the numerical output for different robot models and different inputs (e.g. the joint status $\mathbf{q}$, $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$). For example we used the MATLAB code available on Roy Featherstone's web page about spatial vector algebra and dynamics algorithms (Featherstone 2013), and also the SL software package, which has been in development for more than fifteen years and is used in several research labs for simulations and hard real–time model–based control of real robots (Schaal 2009).

As an additional validation, we partially rewrote a controller for our quadruped robot HyQ using our approach. This program – which is detailed in another paper from our research group (Focchi et al. 2012) – controls the impedance of a single leg of the robot, and exploits coordinate transforms, the Jacobian ${}_{base}J_{foot}$, and so on. We replaced the original implementation of some of these quantities and we managed to obtain the same behavior from the robot. More importantly, once the task was developed, it was a matter of minutes to generate the same expressions for a different leg and have the software control its impedance instead.

7.2.3 Performance comparisons

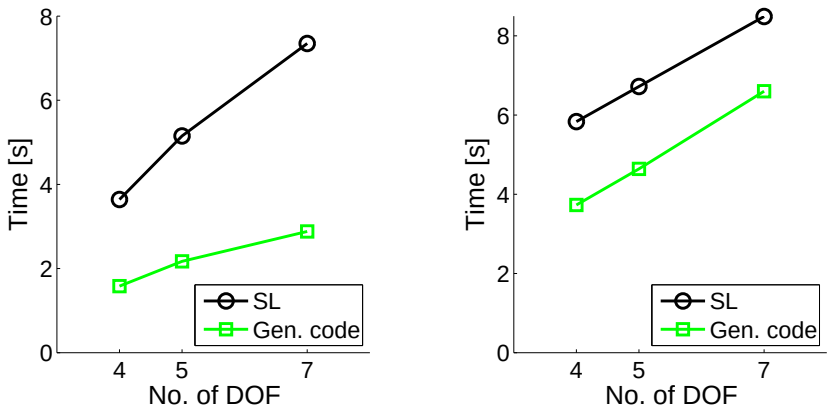
As mentioned in the beginning of this section, we focused on speed comparisons between our generated C++ code (using the Eigen library) and other implementations, to demonstrate that our approach not only provides ease of use and flexibility, but also high run time performances. The execution times shown in the graphs are more relevant as a way of comparing two programs rather than in absolute terms, since they depend also on the specific computer that run them[1].

[1] All the tests were executed on a Intel(R) Core(TM)2 Duo CPU, P8700 @ 2.53GHz, with 4Gb of RAM

However, the absolute execution time and the repeatability of it are also important to show compatibility with real–time loops at frequencies in the order of few hundred Hz, so these aspects will also be addressed in the experiments.

We estimate the execution time of the various functions by means of the standard library function `std::clock()`. Calling repeatedly the same function multiple times (see e.g. the test of Figure 7.6), and possibly averaging the results, helps in smoothing unpredictable time variations not due to the algorithms themselves (e.g. CPU load). However, we are not interested in measurements with millisecond precision, but rather on more significative differences.

We made some comparisons with the SL simulator; as mentioned in Section 3.5, although SL is not optimal as far as usability and flexibility are concerned, it generates a highly optimized low level C code implementation whose performance can very well be considered as a reference. The graph in Figure 7.6



(a) The *Recursive Newton–Euler algorithm* for inverse dynamics

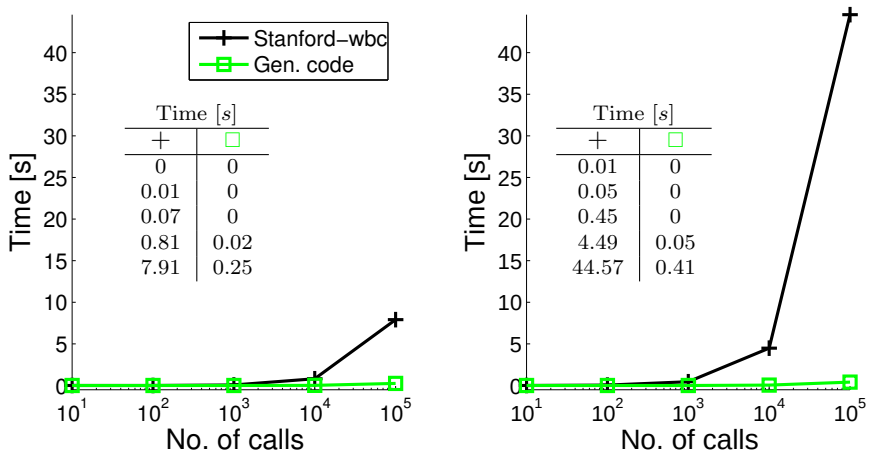
(b) The *Articulated–Body algorithm* for forward dynamics

Figure 7.6 – Performance comparison between the code generated by our framework (C++) and the SL software. Both plots show the cumulative execution time for 10^6 calls of the function (a) $\boldsymbol{\Gamma} = id(\ddot{\mathbf{q}},\ \mathbf{q}, \dot{\mathbf{q}})$ (inverse dynamics) and (b) $\ddot{\mathbf{q}} = fd(\boldsymbol{\Gamma}, \mathbf{q}, \dot{\mathbf{q}})$ (forward dynamics), as a function of the number of degrees of freedom of three robot models.

illustrates some results related to the *Recursive Newton–Euler algorithm* and the *Articulated–Body algorithm* to solve respectively the inverse and the forward dynamics of an articulated robot. The graphs show how the algorithms scale as a function of the number of DoFs, and at the same time they show a speed comparison. We used a four-DoF robot (a leg of HyQ attached to a vertical slider), a five-DoF robot with revolute and prismatic joints and finally a seven-DoF model obtained by adding a two link branch to the previous robot. As can be seen from the plot, the execution times of the two implementations basically have the same order of magnitude, with our implementation being slightly faster. The point of the comparison is that we achieved a performance analogous to a software created to do real–time control more than fifteen years ago (hence extremely concerned about efficiency), but with a process where the

ease of use and the maintainability and observability of the modeling/generation process are greatly improved.

As discussed in Section 6.2.1, we can generate the *Composite–Rigid–Body algorithm* to efficiently compute the joint space inertia matrix $\mathbf{H}$. This algorithm is known to be the most efficient for this job, especially because it exploits a possible sparsity pattern in the matrix. The algorithm requires almost all the spatial force vector transforms in the form ${}_{parent}\mathbf{X}_{child}$ and the motion vector transforms ${}_{child}\mathbf{X}_{parent}$, where *child* and *parent* refer to a pair of connected links, thus its performance depends also on their implementation, which is also taken care of by the code generator.
We performed some comparisons of the execution time with the S-WBC software (see Chapter 3) and the results are summarized in Figure 7.7.
Tests were executed with two different robot models:



Time [s]	
+	□
0	0
0.01	0
0.07	0
0.81	0.02
7.91	0.25

Time [s]	
+	□
0.01	0
0.05	0
0.45	0
4.49	0.05
44.57	0.41

(a) A 5-DOF robot model, with a linear structure. The average time ratio is 36.07.

(b) A 12-DOF quadruped robot. Average time ratio of 99.25.

Figure 7.7 – Performance comparison between the code generated by our framework (black plus) and the *Stanford Whole Body Control* software (green square), for the calculation of the Joint Space Inertia Matrix (both implementations are in the C++ language and use the Eigen library). The x axis represents the number of calls to the function, the y axis the total execution time – also written in the table. The average ratio between the two graphs quantifies roughly the difference in performance.

1. A 5-DOF fictitious robot, composed of a linear kinematic chain with three revolute joints and two prismatic joints, alternated.
2. A 12-DOF robot (our quadruped HyQ), which has four 3-DOF legs and therefore exhibits a branched kinematic structure determining a significant sparsity in its 12×12 inertia matrix.

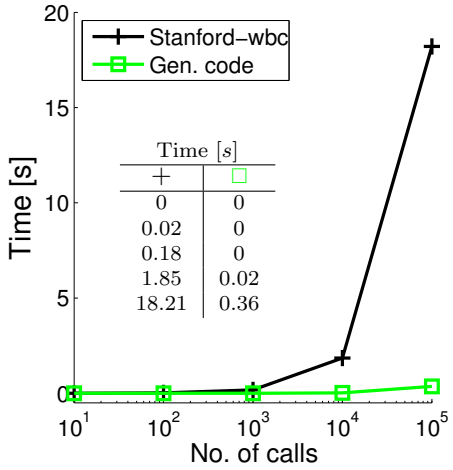
We measured the cumulative execution time of multiple calls to update $\mathbf{H}(\mathbf{q})$, by means of the standard library function `clock()`. For the S-WBC, for instance,

this simply means to measure the single call `computeMassInertia()` of the joint space model class. Even though we do not claim these to be definite and exhaustive comparisons, the plots clearly show a significant faster execution of our implementation, by a factor of about 35 for the first robot.
For the second robot with a branched structure the gain raises dramatically up to 100, most probably because in the S-WBC the sparsity of $\mathbf{H}$ cannot be exploited. In fact, the implementation we tried (version 1.1) is based on a general purpose dynamics engine, and the computation of $\mathbf{H}$ is performed via multiple calls to an inverse dynamics routine (that is a known alternative way to compute $\mathbf{H}$, though not the most efficient). This amounts to computations for each of the 144 elements of the matrix of this example, even though 108 of them are actually zero, which in addition to the general lower speed of the dynamics engine results in a much longer execution time.

Figure 7.8 shows an additional comparison with the S-WBC of the computation time for the null space projector $\mathbf{N}$. This is a common ingredient of the operational space control formulation, used to compute velocities and torques at the joints that do not result in motion at the end effector of the robot. One possible definition is the following:

$$\mathbf{N} = \mathbf{I} - \mathbf{H}^{-1}\,\mathbf{J}^{T}\,\left(\mathbf{J}\,\mathbf{H}^{-1}\,\mathbf{J}^{T}\right)^{-1}\mathbf{J}$$

where all the terms (besides obviously the identity matrix) are function of the joint status $\mathbf{q}$. As you can see, it requires a Jacobian (e.g. the base to end-effector Jacobian) and the inverse of $\mathbf{H}$, so we are basically comparing the computation of both these terms in the S-WBC and in our software (without performing the actual product of all the terms, which would not make the comparison more meaningful).
We performed the test only on the same 5-DOF model of the previous example,



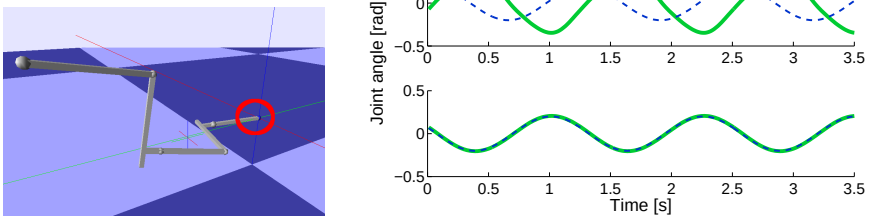
Time [s]	
+	□
0	0
0.02	0
0.18	0
1.85	0.02
18.21	0.36

Figure 7.8 – Performance comparison with the S-WBC of the computation of the inverse of the Joint Space Inertia Matrix and the end–effector Jacobian, for a 5-DOF robot. The x axis represent the number of repeated calls to the appropriate functions, while the y axis is the total execution time. The average ratio between the two plots is 71.54.

not to include the effect of sparsity since we know already how significantly it affects the gap between the S-WBC and the generated code. We selected a hypothetical end–effector at the tip of the last link, and used the corresponding Jacobian.
Also for this case the plot demonstrates a better performance for the code generated with our approach. The average ratio of the execution time is even higher than before (approximately 70 versus 35), as one might have expected because we now have two terms computed possibly in a more efficient way. We get $\mathbf{H}^{-1}$ by computing the factorization $\mathbf{H} = \mathbf{L}^T\mathbf{L}$ and then the product $\mathbf{L}^{-1}\mathbf{L}^{-T} = \mathbf{H}^{-1}$; these steps are also implemented by generated code exploiting the robot–specific sparsity (if any – see Section 6.2.1). These operations obviously add a cost to the computation of $\mathbf{H}$, which however seems less than the increase in execution time observed in the S-WBC to compute $\mathbf{H}^{-1}$ with respect to $\mathbf{H}$; for these tasks, the current implementation of the S-WBC uses a call respectively to the forward dynamics and the inverse dynamics routines.
The computation of the Jacobian is also relevant in determining the performance: our implementation is highly optimized since it has been generated in advance for a specific, known point of the kinematic tree, and therefore outperforms a regular, generic implementation that relies only on computations at run–time.

7.2.4 Simulation and control

Figure 7.9 refers to a simulation we performed using the generated code for the *Articulated–Body algorithm*. In particular, we modified the existing implementation of the SL simulator (Schaal 2009) to replace its dynamics engine with the code generated by our framework. We could then run the program without



(a) Screenshot of the graphic interface. The red circle highlights the first (revolute) joint of the model.

(b) Position tracking at the first joint, without and with inverse dynamics (respectively top and bottom plots). The solid green line (—) is the actual status of the joint, while the dashed blue (- -) is the reference trajectory.

Figure 7.9 – A simulation with SL, modified to use the generated code for the forward dynamics. The simulated robot is a fictitious 5-DOF robot (the same model used in some of the tests shown above), visible in (a). The simulated behavior consists in a sinusoidal trajectory at each joint; the plots in (b) show the position tracking at the first joint, and the improvement of the performance when inverse dynamics is used. Also the inverse dynamics terms in this experiment are computed by our generated code.

additional modifications, and simulate the motion of the robot with any desired trajectory, for example a sinusoidal trajectory at each joint. The behavior of the simulator was indistinguishable from the one obtained with the original version of the program. This exeperiment provides a strong evidence about the correcteness of our implementation of the forward dynamics, and also shows that our C++ code is easily reusable. The steps required to replace the dynamics engine are limited to replacing the call to the forward dynamics function, after properly copying the joints status variables in the appropriate structure, and adding an additional library to the building system.
In fact, we also replaced the call to the inverse dynamics function, which is used by the controller running within SL. In this way all the dynamics computations performed by SL were actually replaced by our generated code. The example of Figure 7.9 shows the tracking of the position of a joint of the robot, that dramatically improves when inverse dynamics is turned on (see Section 2.2).

The execution times illustrated in the previous section and the knowledge that the generated C++ code basically contains only numerical operations performed on memory allocated at initialization time, already provide significant evidence of the compatibility with real–time execution (see Section 4.2.1).
However, to provide a more rigorous argument for this claim, we performed some tests where some of the generated code is running within the control processes of actual robots, in a real–time environment. Figure 7.10 contains a few snapshots of an experiment run on the hydraulic leg of HyQ that consists in tracking a very fast (5 Hz) sinusoidal trajectory at the joints. The desired mo-

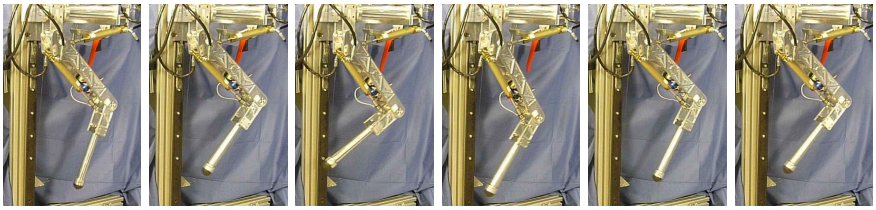


Figure 7.10 – An experiment with the hydraulic leg of HyQ using the generated C++ code for inverse dynamics in the control loop. The task of the robot is to track a sinusoidal trajectory at a frequency of 5 Hz at both joints. This test proved the full compatibility of the generated code with the constraints of real–time execution at 250 Hz.

tion commanded to the robot is so fast that proper position tracking is only possible using an inverse dynamics based controller, which takes advantage of the knowledge about the dynamics of the robot to compute more effectively the commands for the actuators (see Section 2.2 and 5.1). This experiment was in fact originally designed to demonstrate the effectiveness of a force controller based on a hydraulic model, since a proper force control is required to use inverse dynamics (Boaventura, Semini, Buchli, and Caldwell 2011).
In this test we modified the control software based on SL and on our hardware I/O components, by plugging in the implementation of inverse dynamics generated for the model of the leg. This part of the software ran at 250 Hz without

issues, especially without violating the real–time guarantees.[2]

[2]In a Xenomai based real–time environment such as the one we use on our robots, this is certified by the absence of the so–called *mode switches*; these events happen whenever a real–time process executes an illegal operation that would violate the real–time constraints, forcing the real–time operating system to downgrade it temporarily into the non–real–time class.

Chapter 8

Conclusions and future work

Developing robotics software is a challenging activity because of a variety of tasks that have to be performed in a well orchestrated manner. Conflicting requirements such as efficiency and modularity make the problem even harder, together with the inherent complexity of certain functions, such as those related to rigid body dynamics. The lack of general models and implementations often results in poor or missing design and high integration costs; sometimes the reason is that a certain problem is still subject of research and hence not yet fully understood, but often it is simply because research explicitly on robotics software has not been pursued as much as research on control, actuation, mechanical design (which are obviously equally fundamental for a progress in the field).

The software system currently controlling the HyQ robot (Chapter 4) enabled to start a promising research. Different behaviors in the form of generators for the desired joint positions and forces can be developed and tried on the robot. The results of this work have been presented in Section 7.1; they demonstrate that the existing software satisfies the strict requirements of hard real–time control and also exhibits a good degree of flexibility and generality, with respect to changes in the actual robot hardware.
However, such a system is not the definite solution to actually enable versatility and autonomy for the HyQ robot or other machines of similar complexity; some limitations and a direction for future development are discussed below in Section 8.1.

Motivated by the importance of model–based control for articulated robots, the thesis then shifts its focus on software for the kinematics and the dynamics of this class of robots. This particular domain includes some common, recurring problems that as such should be solved by principled and general approaches, to free resources for the required exploratory sides of robotics research.

We have designed a few Domain Specific Languages for the specification of the kinematics and dynamics models for articulated robots, and of algebraic quantities like coordinate transforms and Jacobians, which are ubiquitous in

robot controllers.
The DSLs are based on general domain models that capture the relevant information of the problem, such as the parameters required to fully specify the physics of a multibody system. By using this information it is possible to generate executable code, for instance rigid body dynamics algorithms; such code is efficient and compatible with real–time constraints at high frequencies, e.g. in low level control loops. Code in different programming languages can be generated, addressing different deployment scenarios (e.g. control of a real robot and simulations).

A general aim of our efforts is to relieve roboticists from spending time on non–problems, which are issues well understood in theory and therefore often of scarce scientific interest, but yet critical for successful robotics applications. Effective software solutions for these (non-)problems can still be demanding, and contribute significantly to the costs of development and especially maintenance of the system. The code generation relieves researchers from manually coding critical and complex parts of the software, as well as parts that would just be time consuming and error prone.

Our approach has proved to be effective, in that it exhibits a diversity of desirable features for robotics software: it is easy to use because the user only deals with high level information; the use of sound domain models improves the observability of the process and documentation in general; it is robust and limits human mistakes because it is based on automatic code generation; yet it is effective for real–time robot control, because the generated implementation is fast and efficient.

8.1 A software architecture for the HyQ robot

Achieving sophisticated behaviors with a high degree of autonomy in articulated robots is a challenging objective. Legged locomotion on rough terrain is a notable example of such behaviors – and it is also the main topic of research in the HyQ project, currently under development at our lab (Boaventura, Semini, Buchli, Frigerio, et al. 2012; Focchi et al. 2012; Semini, Tsagarakis, et al. 2011).

For such applications, the architecture described in Chapter 4 has some limitations, and the required improvements will be the subject of future research. Two main limitations that can be immediately identified are:

- A single-process access to hardware resources (as the motor control process using the I/O library) does not handle well different kind of information flows, which typically differ in the amount of data and the frequency. There is no point, for example, of continuously sampling an Inertial Measurement Unit at the same high frequency which is instead required for position sensors, to make the controller work. With a stereo camera emitting depth maps, it is not only inappropriate, but also unfeasible.

- The robot behavior module can be arbitrarily complicated, and as such it requires dedicated analysis and design phase. A single function implementing some logic to generate joint trajectories is not sufficient. This part of the system depends significantly on the specific application.

In general, it is necessary to devise a proper, sound software architecture for the autonomous control of an articulated robot. This activity implies identify-

ing and quantifying the information flows and the logical activities the system has to perform, according to the domain requirements. A logical architecture of the system must be designed with absolutely no reference to any implementation technology, which has to be chosen in a subsequent step (Bernini et al. 2010). As a matter of fact, sometimes in the robotics community the availability of libraries, components, middlewares that might have gained a certain fame (sometimes because they are actually effective), leads to technology driven systems. This is a fundamentally flawed approach, since the technologies should be chosen afterwards, among those matching the requirements of the design (even though it is still rather unlikely for the analysis-design-implementation process to be purely linear and sequential).

The system described in Chapter 4 also describes the deployment of the software on a single computer that acts as the "brain" of the HyQ robot. The use of a single computer is also a possible limitation, since the code for weakly coupled activities or with very different requirements (e.g. low level, hard real–time position control versus the processing of camera images) might very well be more effective if deployed on different machines.
It is true though, that often in robotics there is not really much choice about the hardware, because of other strict constraints that are not arising from the problem domain: room and power availability on the robot, costs associated to the development and maintenance on different platforms – something that can significantly affect how the project progresses. However, any known limitations of this kind should not influence the first design of the architecture; the possible adaptations required to fit the system within any hardware constraint should be made afterwards, with the ideal, purely problem-driven architecture available as a reference.[1]

The actual problem of making a robot such as HyQ capable of moving autonomously in unstructured environment, requires a significant amount of analysis about various issues, to devise a coherent architecture. As possible directions for future work, we can briefly list here some topics that are likely to require some investigation:

- Terrain and world modelling: sophisticated sensors such as cameras and laser scanners are useless if their data is not used to construct some sort of description of the terrain and the environment the robot has to move in. Such a description must be formalized.
- Robot task model: a description of the possible high level commands for the robot should be identified, e.g. how to ask the robot to reach a certain position, with a certain gait and/or a certain velocity, with time constraints, etc.
- Robot model: the control logic could exploit information like the current gait and speed of the robot, the power availability, the estimated quality of the various sensor measurements.

The remark about the use of dynamics simulations on the robot itself – mentioned earlier in Section 5.1.2.1 – also raises some interesting points related to

[1] In this case we are referring primarily to the deployment architecture, which specifies the technologies and the hardware platforms best suited to implement and run the logical components identified in previous stages of the design (Bernini et al. 2010).

the design of the architecture, such as the representation of time and of the own computing capabilities of the robot (a sort of *Quality of Service* measurement); for example, one can imagine a quadruped robot that while running perceives a moving and approaching obstacle (such as a car): planning a behavior that would avoid crashing with the obstacle involves reasoning about its speed and the speed of the robot (which pertains the *real* physical time of the events in the environment) but possibly taking into account how long it would take to do the reasoning itself, that is how long the robot can afford to "think" before making a decision.

8.2 The robotics code generator

8.2.1 Discussion

The results presented in Chapter 7 provide substantial evidence that our approach and the resulting code generation framework are effective, and can be used to aid the development of simulators and controllers. Naturally, many improvements and developments are now possible, and this section tries to summarize some of them as well as provide some hints for discussion.

We met the project goal of building the framework with open source technologies only, increasing the chances that the tool gets adopted by a wide community that would then provide useful feedback. A possible release of our source code with a similar license would also enable a contribution from the community at the development level.
It is however crucial to avoid the proliferation of features that would not be coherent with the existing models, as well as polluting the same models with concepts belonging in fact to other domains (e.g. adding elements related to a possible graphical interface in the kinematics DSL). It can indeed be tempting to incrementally add features to the existing software just by following the needs of the community (which are obviously not limited to software for kinematics and dynamics), without careful analysis of the new issues.

Springs in the actuators or in the joints of the robot is a typical example, since their usage has become quite common with the recent trends in robotics research (Hutter et al. 2012; Tsagarakis et al. 2013), so that appropriate software solutions related to such mechanisms are required. All our framework is based on the Rigid Body Dynamics model, that does not deal with any dynamics of the actuation and assumes ideal force sources at the joints. Therefore topics like Series Elastic Actuators shall not be introduced in the current framework. It is actually an interesting point – related to both control theory and software – whether it is possible for such mechanisms to be controlled in such a way that higher level software can abstract them as force sources, or, on the contrary, which information about their peculiar dynamics have to flow across different software layers.
Springs in parallel to actuators, on the other hand, typically result simply in additional forces acting on the links of the robot. Dynamics algorithms such as those described in Section 5.1.2.2 already handle the case of additional external forces acting on the rigid bodies, so no particular development of the framework seems necessary.

As explained earlier in the text (e.g. Section 7.2) the C++ code generated with our framework is based on the Eigen library for linear algebra. The injection of such a dependency in the generated code, especially in the targets that can be used in real–time controllers, may be seen in contrast to the idea of having full control over the code running on a real robot, which is also one of the motivations for our framework (e.g. as opposed to relying on some general purpose dynamics engine that may have other dependencies, may not be real–time compatible, etc.). However we believe this is not a severe limitation, since Eigen is a carefully designed library, focused on performance, which has been shown in our results to be appropriate for real time; in other words, Eigen seems to be a good tool for the job, worth the minor drawback of having a dependency.
If properly used (e.g. no dynamically sized matrices) Eigen leads to compiled code that is likely to be faster or as fast as the code that would be generated by expanding all the vector operations into scalar ones, i.e. products and sums addressing the elements of the matrices one by one. This approach is possible but results in a much more complicated generation software as well as much more complicated generated code; another drawback is that non-vector operations cannot exploit possible hardware specialized for vector arithmetic (Featherstone 2008) (Eigen *does* perform optimizations based on the target architecture).
Note that similar facilities as those provided by Eigen in C++ might not be available in other programming languages. In such cases the transformation of all the algebra into scalar operations would be mandatory.

Another remark about efficiency and symbolic generation concerns sparsity and sparse matrix implementations. The use of sparse matrices in the generated code might make generic computations (sums and products in user code) faster. In principle, given a known matrix (say, a coordinate transform), the generator could choose whether or not to use a sparse matrix implementation in the target language when generating code. However it is quite difficult to devise such criteria in general (when is it worth to use a sparse matrix? For which degree of sparsity?) also because the actual benefits highly depend on the specific implementation.
In fact literature refers to the use of symbolic engines to identify the multiplications by zero to avoid generating code for such operations (Featherstone 2008). This would require to transform vector arithmetic in scalar operations or to rely on a sparse matrix implementation taking care of the zeros, both points just discussed above.
Note that this approach is different from the optimizations we have described in Section 6.2.1. Our code generators take advantage of prior information about the kinematics of the robot so that they know *in advance* that certain numbers will be zero, without relying on a symbolic engine. Refer to the explanations and the examples in Section 6.2.1, about the motion subspace matrix and the branch–induced sparsity.
In our framework, element–wise optimizations take place for coordinate transforms and Jacobians, in that elements recognized as constant are not recomputed each time the matrix is updated with a new joint status vector $\mathbf{q}$ (cf. Section 6.2.2.3). On the other hand, the *products* having these matrices as operands are not explicitly optimized, even if they contain a few zeros. Drawbacks about optimizations based on null elements have just been described. Whether it is still worth doing them must be evaluated with experiments, and may be a topic

for future research.

Another topic worth mentioning in this discussion concerns the flexibility of our framework. One of the strong motivations behind the use of a code generation system is that it provides efficient implementations without sacrificing generality. In fact some of the flexibility available with general purpose libraries is lost, but it is quite negligible since it is limited to very unlikely use cases. For example, with our approach it is not possible to *dynamically* (i.e. at robot-run-time) change the dynamic model being used for control/simulations. But the chances that a robot changes its structure while active on the field are quite limited (unless one is dealing with modular and reconfigurable robots (Schultz et al. 2007)). On the other hand, the generated code can easily deal with the much more likely case of a change in the mass or the mass distribution, e.g. if new payload is being carried: simply, new parameters have to be loaded (the generated code should not have numerical constants spread all over the sources, even if automatically generated, exactly for these reasons).
Another argument one may raise is about Jacobians. It is obvious that we cannot generate the code for Jacobians for any arbitrary point determined at run–time. While it is common that some points of interests on the robot are known or can be estimated in advance (e.g. the end-effectors), it is true that other points depend on run–time operation (e.g. contact points). However, our framework already provides a robust way to address also these issues: as an example, one might generate the Jacobians for every link of the robot, and pick one of them at run–time once a specific point on the same link is identified; then the link Jacobian can be used for velocity or force computations simply by computing the appropriate transform that takes care of the distance of the point from the default link frame (to which the link Jacobian refers to).

In conclusion to this point about flexibility, note that in scenarios where the robot control system is at an advanced level, it is definitively plausible to imagine the code generation facilities themselves as part of the software of the robot. In certain conditions the robot may compute different models than those it is currently embedding, stop, generate code out of them, and then start using the fresh new code, when the speed of execution is a critical issue.
Stopping and reasoning as a reaction of a difficult situation or an unforeseen event is a totally sensible approach, adopted by humans as well.

8.2.2 Future improvements

In general, various improvements and further features can be added to the existing code generation framework, also because it is not only meant to be a proof of concept of basic research, but it aims at being concretely a useful tool for roboticists. Below is a list of some possible future developments:

- Add to the software models the explicit notion of *task* and *task space*. In this way the software could reason about properties such as the dimension of a task space, for example to generate a Jacobian with the significant rows only, to automatically detect task–specific redundancy, or to identify the subset of joints involved in a specific task.

- Additional classes like `Chain` and `Tree`, explicitly modelling sub-parts of the whole robot assembly, should be added. This point is related to the

previous one, since particular task spaces may be associated to specific sub-trees of the robot (e.g. the positioning of a hand of a humanoid and the corresponding arm).

- Perform more experiments with real robots using the generated code. Compare the performances of the C++ code with additional existing tools, such as Robotran and SD/Fast.
- Add other targets for the code generation, addressing for instance algorithms for floating base robots, and the articulated body algorithm for the forward dynamics problem.
- Extend the models and the code generation targets to support closed loop systems. As an example, the notion of *loop joint* should be added to the class diagram of Figure 5.6 noting that a distinctive feature of this kind of joints is that they do not induce a parent-child relationship, and they solely constitute additional kinematic constraints.
- Deal explicitly in the framework with 3-DoF spherical joints, to avoid numerical singularities that may arise when modelling them with a sequence of three simple revolute joints.
- Include in the kinematics DSL elements about the range of motion of the joints, which is not relevant for dynamics algorithms but it is definitely part of a description of the kinematics of a robot.
- Improve the validation of the DSL documents with checks of semantic constraints which cannot be enforced by the grammar – e.g. a link cannot be the child of more than one other link; the inertia tensor of each link must be a positive definite matrix.
- Devise the relation between the joint parameters in the kinematics DSL (see Section 5.2.1.1) and the Denavit–Hartenberg parameters. A clear and robust conversion routine between the two conventions would facilitate the use of our software within the community, since roboticists tend to be familiar with DH parameters, and the specifications of many robots provide only them. Note that DH parameters can represent only a subset of configurations with respect to the six parameters of the kinematics DSL (Featherstone 2008); therefore the conversion *to* DH parameters might not always be possible.
- Further optimizations. Currently, in the generated C++ code, the update of a transform or a Jacobian computes only once a set of unique trigonometric functions ($sin(q_1)$, $cos(q_3)$, etc.), even if multiple elements of the matrix depend on the same ones. This kind of optimization is not yet happening at the global level, among different matrices that might be function of some shared terms (e.g. ${}_{base}\mathbf{X}_{link1}$, ${}_{base}\mathbf{X}_{link2}$, ${}_{base}\mathbf{X}_{link3}$, etc.). Realizing this improvement is mainly a matter of tuning the existing implementation; it may also require enriching explicitly the representation of a transform with the kinematic chain it refers to, to detect overlaps.
- Make the software more configurable by the user, e.g. by means of configuration files. It would then be possible to further customize the generated code, and better suit the needs or the taste of the users.

- Address other programming languages, such as Java or Python.

Appendix A

Generated code example

This section shows an example of the C++ code generated by our framework, to give a general idea of how the output code currently looks like. The example refers to the implementation of the *Recursive Newton–Euler algorithm* for inverse dynamics, for the robot HyL (a single leg of HyQ, attached to a vertical slider).

Currently, code is generated in the form of a class with a few methods, which implement the full algorithm or simplified versions to compute e.g. only the gravity compensation terms. The namespace `iit::rbd` contains some utility defintions, like a type representing a 6D force vector. These definitions are provided by a few static (i.e. non–generated) header files that are available for download together with the framework. Besides these headers, all the generated code depends solely on the Eigen library for linear algebra (Guennebaud et al. 2013).

For more information, please refer to:

- Figure 6.10 and 7.10 about the HyL robot.
- Section 5.1.2.2 and 6.2.1 about dynamics algorithms and code generation.
- Chapter 5 of (Featherstone 2008) for an in–depth description of the algorithm.

A.1 Header file

```
#ifndef IIT_HYL_INVERSE_DYNAMICS_H_
#define IIT_HYL_INVERSE_DYNAMICS_H_

#include <Eigen/Dense>
#include <iit/rbd/rbd.h>
#include <iit/rbd/InertiaMatrix.h>
#include <iit/rbd/utils.h>

#include "declarations.h"
#include "transforms.h"
#include "link_data_map.h"

namespace iit {
namespace HyL {
```

```
namespace dyn {

typedef iit::rbd::InertiaMatrixDense InertiaMatrix;
typedef LinkDataMap<iit::rbd::ForceVector> ExtForces;

/**
 * The Inverse Dynamics routine for the robot HyL.
 *
 * In addition to the full Newton-Euler algorithm, specialized versions to
 * compute only certain terms are provided.
 * The parameters common to most of the methods are the joint status \c q, the
 * joint velocities \c qd and the accelerations \c qdd. The \c torques parameter
 * will be filled with the computed values.
 * Additional overloaded methods are provided without the \c q parameter.
 * These methods use the current configuration of the robot; they are provided
 * for the sake of efficiency, in case the kinematics transforms of the robot
 * have already been updated elsewhere with the most recent configuration (eg
 * by a call to setJointStatus()), so that it is useless to compute them again.
 */
class InverseDynamics {
public:
    InverseDynamics();
    /** \name Inverse dynamics
     * The full Newton-Euler algorithm for inverse dynamics
     */ ///@{
    void id(const JointState& q, const JointState& qd, const JointState& qdd,
            JointState& torques);
    void id(const JointState& q, const JointState& qd, const JointState& qdd,
           const ExtForces& fext, JointState& torques);
    ///@}
    /** \name Gravity terms
     * The torques acting on the joints due to gravity, for a specific
     * configuration. In order to do gravity compensation, torques with the
     * opposite sign should be applied.
     */ ///@{
    void G_terms(const JointState& q, JointState& torques);
    void G_terms(JointState& torques);
    ///@}
    /** \name Centrifugal and Coriolis terms
     * The torques acting on the joints due to centrifugal and Coriolis effects.
     */ ///@{
    void C_terms(const JointState& q, const JointState& qd,
                 JointState& torques);
    void C_terms(const JointState& qd, JointState& torques);
    ///@}
    /** Updates all the kinematics transforms. */
    void setJointStatus(const JointState& q) const;
public:
    iit::rbd::SparseColumnd gravity;
protected:
    void firstPass(const JointState& q, const JointState& qd,
                   const JointState& qdd);
    void secondPass(JointState& torques);

private:
    iit::rbd::Matrix66d spareMx; // support variable
    // Link 'slider' :
    InertiaMatrix slider_Imx;
    iit::rbd::VelocityVector slider_v;
    iit::rbd::VelocityVector slider_a;
    iit::rbd::ForceVector slider_f;
    // Link 'hip' :
```

```
    InertiaMatrix hip_Imx;
    iit::rbd::VelocityVector hip_v;
    iit::rbd::VelocityVector hip_a;
    iit::rbd::ForceVector hip_f;
    // Link 'leg' :
    InertiaMatrix leg_Imx;
    iit::rbd::VelocityVector leg_v;
    iit::rbd::VelocityVector leg_a;
    iit::rbd::ForceVector leg_f;
    // Link 'lowerleg' :
    InertiaMatrix lowerleg_Imx;
    iit::rbd::VelocityVector lowerleg_v;
    iit::rbd::VelocityVector lowerleg_a;
    iit::rbd::ForceVector lowerleg_f;

};

inline void InverseDynamics::setJointStatus(
        const JointState& q) const
{
    transforms6D::fr_slider_X_fr_base(q);
    transforms6D::fr_hip_X_fr_slider(q);
    transforms6D::fr_leg_X_fr_hip(q);
    transforms6D::fr_lowerleg_X_fr_leg(q);
}

}}}
#endif
```

A.2 Definitions file

```
#include "inverse_dynamics.h"
#include "inertia_params.h"

using namespace std;
using namespace iit::rbd;
using namespace iit::HyL::dyn;

iit::HyL::dyn::InverseDynamics::InverseDynamics() {
    gravity.resize(6);
    gravity.insert(5) = 9.81;

    slider_v.setZero();
    hip_v.setZero();
    leg_v.setZero();
    lowerleg_v.setZero();

    InertiaParameters linkInertias;
    slider_Imx = linkInertias.getTensor_slider();
    hip_Imx = linkInertias.getTensor_hip();
    leg_Imx = linkInertias.getTensor_leg();
    lowerleg_Imx = linkInertias.getTensor_lowerleg();

    transforms6D::initAll(); // initializes coordinates transforms
}


void iit::HyL::dyn::InverseDynamics::id(
        const JointState& q, const JointState& qd,
        const JointState& qdd, JointState& torques)
{
    transforms6D::fr_slider_X_fr_base(q);
```

```
    transforms6D::fr_hip_X_fr_slider(q);
    transforms6D::fr_leg_X_fr_hip(q);
    transforms6D::fr_lowerleg_X_fr_leg(q);
    firstPass(q, qd, qdd);
    secondPass(torques);
}
/**
 * \param fext the external forces acting on the links. Each external force
 * must be expressed in the frame of the link it is exerted on.
 */
void iit::HyL::dyn::InverseDynamics::id(const JointState& q,
        const JointState& qd, const JointState& qdd,
        const ExtForces& fext, JointState& torques)
{
    transforms6D::fr_slider_X_fr_base(q);
    transforms6D::fr_hip_X_fr_slider(q);
    transforms6D::fr_leg_X_fr_hip(q);
    transforms6D::fr_lowerleg_X_fr_leg(q);
    firstPass(q, qd, qdd);
    // Add the external forces:
    slider_f -= fext[SLIDER];
    hip_f -= fext[HIP];
    leg_f -= fext[LEG];
    lowerleg_f -= fext[LOWERLEG];
    secondPass(torques);
}

void iit::HyL::dyn::InverseDynamics::G_terms(
        const JointState& q, JointState& torques)
{
    transforms6D::fr_slider_X_fr_base(q);
    transforms6D::fr_hip_X_fr_slider(q);
    transforms6D::fr_leg_X_fr_hip(q);
    transforms6D::fr_lowerleg_X_fr_leg(q);
    G_terms(torques);
}

void iit::HyL::dyn::InverseDynamics::G_terms(JointState& torques) {
    // Link 'slider'
    slider_a = transforms6D::fr_slider_X_fr_base.col(5)*(-iit::rbd::g);
    slider_f = slider_Imx * slider_a;
    // Link 'hip'
    hip_a = (transforms6D::fr_hip_X_fr_slider * slider_a);
    hip_f = hip_Imx * hip_a;
    // Link 'leg'
    leg_a = (transforms6D::fr_leg_X_fr_hip * hip_a);
    leg_f = leg_Imx * leg_a;
    // Link 'lowerleg'
    lowerleg_a = (transforms6D::fr_lowerleg_X_fr_leg * leg_a);
    lowerleg_f = lowerleg_Imx * lowerleg_a;
    secondPass(torques);
}

void iit::HyL::dyn::InverseDynamics::C_terms(
        const JointState& q, const JointState& qd,
        JointState& torques)
{
    transforms6D::fr_slider_X_fr_base(q);
    transforms6D::fr_hip_X_fr_slider(q);
    transforms6D::fr_leg_X_fr_hip(q);
    transforms6D::fr_lowerleg_X_fr_leg(q);
    C_terms(qd, torques);
```

```
}

void iit::HyL::dyn::InverseDynamics::C_terms(
       const JointState& qd, JointState& torques)
{
    // Link 'slider'
    // velocity:
    slider_v(5) = qd(0);
    // force:
    Utils::fillAsForceCrossProductMx(slider_v, spareMx);
    slider_f = (spareMx * slider_Imx).col(5) * qd(0);

    // Link 'hip'
    // velocity:
    hip_v = (transforms6D::fr_hip_X_fr_slider * slider_v);
    hip_v(2) += qd(1);
    // acceleration and force:
    Utils::fillAsMotionCrossProductMx(hip_v, spareMx);
    hip_a = (spareMx.col(2) * qd(1));
    hip_f = hip_Imx * hip_a + (-spareMx.transpose() * hip_Imx * hip_v);

    // Link 'leg'
    // velocity:
    leg_v = (transforms6D::fr_leg_X_fr_hip * hip_v);
    leg_v(2) += qd(2);
    // acceleration and force:
    Utils::fillAsMotionCrossProductMx(leg_v, spareMx);
    leg_a = (transforms6D::fr_leg_X_fr_hip * hip_a) + (spareMx.col(2) * qd(2));
    leg_f = leg_Imx * leg_a + (-spareMx.transpose() * leg_Imx * leg_v);

    // Link 'lowerleg'
    // velocity:
    lowerleg_v = (transforms6D::fr_lowerleg_X_fr_leg * leg_v);
    lowerleg_v(2) += qd(3);
    // acceleration and force:
    Utils::fillAsMotionCrossProductMx(lowerleg_v, spareMx);
    lowerleg_a = (transforms6D::fr_lowerleg_X_fr_leg * leg_a) +
                  (spareMx.col(2) * qd(3));
    lowerleg_f = lowerleg_Imx * lowerleg_a +
           (-spareMx.transpose() * lowerleg_Imx * lowerleg_v);

    secondPass(torques);
}

void iit::HyL::dyn::InverseDynamics::firstPass(const JointState& q,
       const JointState& qd, const JointState& qdd)
{
    // First pass, link 'slider'
    slider_v(5) = qd(0);
    slider_a = (transforms6D::fr_slider_X_fr_base * gravity);
    slider_a(5) += qdd(0);
    Utils::fillAsForceCrossProductMx(slider_v, spareMx);
    slider_f = slider_Imx * slider_a + ((spareMx * slider_Imx).col(5) * qd(0));
    // First pass, link 'hip'
    hip_v = (transforms6D::fr_hip_X_fr_slider * slider_v);
    hip_v(2) += qd(1);

    Utils::fillAsMotionCrossProductMx(hip_v, spareMx);

    hip_a = (transforms6D::fr_hip_X_fr_slider * slider_a) +
           (spareMx.col(2) * qd(1));
    hip_a(2) += qdd(1);
```

```
    hip_f = hip_Imx * hip_a + (-spareMx.transpose() * hip_Imx * hip_v);
    // First pass, link 'leg'
    leg_v = (transforms6D::fr_leg_X_fr_hip * hip_v);
    leg_v(2) += qd(2);

    Utils::fillAsMotionCrossProductMx(leg_v, spareMx);

    leg_a = (transforms6D::fr_leg_X_fr_hip * hip_a) + (spareMx.col(2) * qd(2));
    leg_a(2) += qdd(2);

    leg_f = leg_Imx * leg_a + (-spareMx.transpose() * leg_Imx * leg_v);
    // First pass, link 'lowerleg'
    lowerleg_v = (transforms6D::fr_lowerleg_X_fr_leg * leg_v);
    lowerleg_v(2) += qd(3);

    Utils::fillAsMotionCrossProductMx(lowerleg_v, spareMx);

    lowerleg_a = (transforms6D::fr_lowerleg_X_fr_leg * leg_a) +
           (spareMx.col(2) * qd(3));
    lowerleg_a(2) += qdd(3);

    lowerleg_f = lowerleg_Imx * lowerleg_a +
           (-spareMx.transpose() * lowerleg_Imx * lowerleg_v);
}

void iit::HyL::dyn::InverseDynamics::secondPass(JointState& torques) {
    // Link 'lowerleg'
    torques(3) = lowerleg_f(2);
    leg_f = leg_f + transforms6D::fr_lowerleg_X_fr_leg.transpose() * lowerleg_f;
    // Link 'leg'
    torques(2) = leg_f(2);
    hip_f = hip_f + transforms6D::fr_leg_X_fr_hip.transpose() * leg_f;
    // Link 'hip'
    torques(1) = hip_f(2);
    slider_f = slider_f + transforms6D::fr_hip_X_fr_slider.transpose() * hip_f;
    // Link 'slider'
    torques(0) = slider_f(5);
}
```

Appendix B

Publications

- Marco Frigerio, Jonas Buchli, and Darwin G. Caldwell (2012a). "Code Generation of Algebraic Quantities for Robot Controllers". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*

- Marco Frigerio, Jonas Buchli, and Darwin G. Caldwell (May 2012b). "Model based code generation for kinematics and dynamics computations in robot controllers". In: *Seventh workshop on Software Development and Integration in Robotics (ICRA SDIR VII)* (extended abstract)

- Marco Frigerio, Jonas Buchli, and Darwin G. Caldwell (Sept. 2011). "A Domain Specific Language for kinematic models and fast implementations of robot dynamics algorithms". In: *2nd International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob)*

The software described in the present thesis and in the papers just cited, can be downloaded from the web at the following address (last checked on March the 17^{th}, 2013):
www.iit.it/en/article/10-advanced-robotics/1253-robotics-code-generator.html
Should the URL above not work anymore, please use a search engine with the following keywords: *frigerio, iit, robotics code generator*.

Ohter publications:

- Victor Barasuol, Jonas Buchli, Claudio Semini, Marco Frigerio, Edson R. De Pieri, and Darwin G. Caldwell (2013). "A Reactive Controller Framework for Quadrupedal Locomotion on Challenging Terrain". In: *IEEE International Conference on Robotics and Automation (ICRA)*. [accepted for publication]

- Thiago Boaventura, Michele Focchi, Marco Frigerio, Jonas Buchli, Claudio Semini, Gustavo A. Medrano-Cerda, and Darwin G. Caldwell (2012). "On the role of load motion compensation in high-performance force control". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*

- Claudio Semini, Hamza Khan, Marco Frigerio, Thiago Boaventura, Michele Focchi, Jonas Buchli, and Darwin G. Caldwell (2012). "Design and Scal-

ing of Versatile Quadruped Robots". In: *Climbing and Walking Robots (CLAWAR)*

- Thiago Boaventura, Claudio Semini, Jonas Buchli, Marco Frigerio, Michele Focchi, and Darwin G. Caldwell (2012). "Dynamic Torque Control of a Hydraulic Quadruped Robot". In: *IEEE International Conference on Robotics and Automation (ICRA)*

- Michele Focchi, Thiago Boaventura, Claudio Semini, Marco Frigerio, Jonas Buchli, and Darwin G. Caldwell (2012). "Torque-control Based Compliant Actuation of a Quadruped Robot". In: *12th IEEE International Workshop on Advanced Motion Control (AMC)*

- Claudio Semini, Jonas Buchli, Marco Frigerio, Thiago Boaventura, Michele Focchi, Emanuele Guglielmino, Ferdinando Cannella, Nikos G. Tsagarakis, and Darwin G. Caldwell (Apr. 2011). "HyQ – A Dynamic Locomotion Research Platform". In: *International Workshop on Bio-Inspired Robots*

References

Barasuol, Victor, Jonas Buchli, Claudio Semini, Marco Frigerio, Edson R. De Pieri, and Darwin G. Caldwell (2013). "A Reactive Controller Framework for Quadrupedal Locomotion on Challenging Terrain". In: *IEEE International Conference on Robotics and Automation (ICRA)*. [accepted for publication] (cit. on p. 83).

Bernini, Diego and Francesco Tisato (2010). "Explaining architectural choices to non-architects". In: *4th European conference on Software architecture*. ECSA'10. Copenhagen, Denmark: Springer-Verlag, pp. 352–359. ISBN: 3-642-15113-2, 978-3-642-15113-2 (cit. on p. 93).

Bischoff, Rainer, Tim Guhl, Erwin Prassler, Walter Nowak, Gerhard Kraetzschmar, Herman Bruyninckx, Peter Soetens, Martin Haegele, Andreas Pott, Peter Breedveld, Jan Broenink, Davide Brugali, and Nicola Tomatis (2010). "BRICS – Best practice in robotics". In: *IFR International Symposium on Robotics (ISR)* (cit. on pp. 4, 7, 21).

Bischoff, Rainer, Johannes Kurth, Guenter Schreiber, Ralf Koeppe, Alin Albu-Schaeffer, Alexander Beyer, Oliver Eiberger, Sami Haddadin, Andreas Stemmer, Gerhard Grunwald, and Gerhard Hirzinger (June 2010). "The KUKA-DLR Lightweight Robot arm – a new reference platform for robotics research and manufacturing". In: *41st International Symposium on Robotics (ISR)*, pp. 1–8 (cit. on pp. 16, 17, 51).

Boaventura, Thiago, Claudio Semini, Jonas Buchli, and Darwin G. Caldwell (2011). "Actively–compliant Leg for Dynamic Locomotion". In: *International Symposium on Adaptive Motion of Animals and Machines (AMAM)* (cit. on pp. 81, 89).

Boaventura, Thiago, Claudio Semini, Jonas Buchli, Marco Frigerio, Michele Focchi, and Darwin G. Caldwell (2012). "Dynamic Torque Control of a Hydraulic Quadruped Robot". In: *IEEE International Conference on Robotics and Automation (ICRA)* (cit. on pp. 11, 19, 40, 81, 92).

Bordignon, Mirko, Ulrik Pagh Schultz, and Kasper Støy (Oct. 2010). "Model-based kinematics generation for modular mechatronic toolkits". In: *ACM SIGPLAN Notices* 46 (2), pp. 157–166. ISSN: 0362-1340 (cit. on p. 21).

Brooks, Alex, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Anders Oreback (Aug. 2005). "Towards component-based robotics". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 163–168 (cit. on p. 22).

Brugali, Davide (2010). "From the Editor-in-Chief: A New Research Community, a New Journal". In: *Software Engineering for Robotics* 1, pp. 1–2 (cit. on p. 21).

CEREM (2013). *Robotran*. Center for Research in Mechatronics (CEREM), iMMC, UCL. URL: www.robotran.be/ (cit. on p. 24).

Corke, Peter (Mar. 1996). "A robotics toolbox for Matlab". In: *IEEE Robotics Automation Magazine* 3.1, pp. 24–32. ISSN: 1070-9932 (cit. on p. 45).

Dijkstra, Edsger Wybe (1982). "On the role of scientific thought". In: *Selected Writings on Computing: A Personal Perspective*, pp. 60–66 (cit. on p. 35).

Efftinge, Sven et al. (2013). *Xtext*. URL: www.eclipse.org/Xtext/ (cit. on pp. 59, 62).

Efftinge, Sven and Sebastian Zarnekow (2013). *The Xtend language*. URL: www.eclipse.org/xtend/ (cit. on p. 59).

Eysholdt, Moritz and Heiko Behrens (2010). "Xtext: implement your language faster than the quick and dirty way". In: *ACM international conference companion on Object oriented programming systems languages and applications companion*. SPLASH '10. New York, NY, USA: ACM, pp. 307–309. ISBN: 978-1-4503-0240-1 (cit. on p. 59).

Featherstone, Roy (2005). "Efficient Factorization of the Joint-Space Inertia Matrix for Branched Kinematic Trees". In: *The International Journal of Robotics Research* 24.6, pp. 487–500 (cit. on p. 69).

— (2008). *Rigid Body Dynamics Algorithms*. Springer (cit. on pp. 3, 18–20, 24, 45, 47–50, 52, 54, 68, 69, 95, 97, 99).

— (2010a). "A Beginner's Guide to 6-D Vectors (Part 1)". In: *IEEE Robotics & Automation Magazine* 17.3, pp. 83–94 (cit. on pp. 20, 56).

— (2010b). "A Beginner's Guide to 6-D Vectors (Part 2)". In: *IEEE Robotics & Automation Magazine* 17.4, pp. 88–99 (cit. on pp. 20, 52, 56).

— (2010c). "Exploiting Sparsity in Operational-space Dynamics". In: *The International Journal of Robotics Research* 29.10, pp. 1353–1368 (cit. on p. 69).

— (2013). *Spatial Vectors and Rigid-Body Dynamics*. URL: royfeatherstone.org/spatial/ (cit. on pp. 20, 48, 84).

Focchi, Michele, Thiago Boaventura, Claudio Semini, Marco Frigerio, Jonas Buchli, and Darwin G. Caldwell (2012). "Torque-control Based Compliant Actuation of a Quadruped Robot". In: *12th IEEE International Workshop on Advanced Motion Control (AMC)* (cit. on pp. 11, 84, 92).

Fowler, Martin (2003). *UML distilled*. Ed. by Rumbaugh Booch Jacobson. Addison-Wesley (cit. on p. 53).

— (2010). *Domain-Specific Languages*. Addison-Wesley (cit. on pp. 11, 12, 24).

Gerum, Philippe (Apr. 2004). *Xenomai – Implementing a RTOS emulation framework on GNU/Linux* (cit. on p. 32).

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandrioli (2002). *Fundamentals of Software Engineering*. Ed. by Second. Pearson Prentice Hall (cit. on pp. 20, 70).

Guennebaud, Gaël, Benoît Jacob, et al. (2013). *The Eigen library v3.* URL: eigen.tuxfamily.org (cit. on pp. 84, 99).

Hogan, Neville (1985). "Impedance control: An approach to manipulation: Part II – Implementation". In: *Dynamic Systems, Measurement, and Control* 107, pp. 8–16 (cit. on pp. 2, 47).

Hutter, Marco, Christian Gehring, Michael Bloesch, Mark A. Hoepflinger, C. David Remy, and Roland Siegwart (2012). "StarlETH: A compliant quadrupedal robot for fast, efficient, and versatile locomotion". In: *15th International Conference on Climbing and Walking Robot (CLAWAR)* (cit. on pp. 15, 17, 94).

Khatib, Oussama (1987). "A unified approach for motion and force control of robot manipulators: The operational space formulation". In: *IEEE Journal on Robotics and Automation* 3.1, pp. 43–53 (cit. on pp. 2, 18, 19).

— (Feb. 1995). "Inertial Properties in Robotics Manipulation: An Object-Level Framework". In: *International Journal of Robotics Research* 14.1, pp. 19–36 (cit. on pp. 18, 45, 47).

Khatib, Oussama, Luis Sentis, Jaeheung Park, and James Warren (2004). "Whole-Body Dynamic Behavior and Control of Human-like Robots". In: *International Journal of Humanoid Robotics* 1.1, pp. 29–43 (cit. on p. 18).

Kiszka, Jan (1997). *The Real-Time Driver Model and First Applications* (cit. on p. 34).

Klotzbücher, Markus, Peter Soetens, and Herman Bruyninckx (Nov. 2010). "OROCOS RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages". In: *Internation Workshop on Dynamic Languages (DYROS)* (cit. on p. 21).

Laet, Tinne De, Steven Bellens, Herman Bruyninckx, and Joris De Schutter (2012). "Geometric Relations between Rigid Bodies (Part 2) - From Semantics to Software". In: *IEEE Robotics and Automation Magazine* (cit. on p. 22).

Laet, Tinne De, Steven Bellens, Ruben Smits, Erwin Aertbelien, Herman Bruyninckx, and Joris De Schutter (2012). "Geometric Relations between Rigid Bodies: Semantics for Standardization". In: *IEEE Robotics and Automation Magazine.* accepted for publication (cit. on pp. 8, 22).

LaValle, Steven M. (2006). *Planning algorithms.* Cambridge University Press (cit. on p. 47).

Mattingley, Jacob and Stephen Boyd (Mar. 2012). "CVXGEN: a code generator for embedded convex optimization". English. In: *Optimization and Engineering* 13.1 (1), pp. 1–27. ISSN: 1389-4420 (cit. on pp. 25, 26).

Maxima (2011). *Maxima, a Computer Algebra System. Version 5.25.1.* URL: maxima.sourceforge.net/ (cit. on pp. 59, 72).

McKain, David (2013). *Jacomax, the Java wrapper for Maxima.* URL: www.wiki.ed.ac.uk/display/Physics/Jacomax (cit. on p. 72).

Mernik, Marjan, Jan Heering, and Anthony M. Sloane (Dec. 2005). "When and how to develop domain-specific languages". In: *ACM Computing Surveys* 37.4 (4), pp. 316–344. ISSN: 0360-0300 (cit. on p. 11).

Meyer, Bertrand (1997). *Object-oriented software construction*. Prentice Hall (cit. on p. 36).

Mistry, Michael, Jonas Buchli, and Stefan Schaal (2010). "Inverse dynamics control of floating base systems using orthogonal decomposition." In: *ICRA*. IEEE, pp. 3406–3412 (cit. on pp. 11, 20).

Mistry, Michael, Jun Nakanishi, Gordon Cheng, and Stefan Schaal (Dec. 2008). "Inverse kinematics with floating base and constraints for full body humanoid robot control". In: *8th IEEE-RAS International Conference on Humanoid Robots*, pp. 22–27 (cit. on p. 20).

Modelica Association (2013). *Modelica and the Modelica Association*. URL: www.modelica.org (cit. on p. 25).

Nakanishi, Jun, Michael Mistry, and Stefan Schaal (2007). "Inverse dynamics control with floating base and constraints". In: *International Conference on Robotics and Automation (ICRA)*, pp. 1942–1947 (cit. on p. 19).

Nayar, Hari D. and Issa A.D. Nesnas (Nov. 2007). "Re-usable kinematic models and algorithms for manipulators and vehicles". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 833–838 (cit. on p. 22).

OpenHRP group (2013). *OpenHRP*. URL: www.openrtp.jp/openhrp3/en/index.html (cit. on p. 25).

Philippsen, Roland, Luis Sentis, and Oussama Khatib (Sept. 2011). "An open source extensible software package to create whole-body compliant skills in personal mobile manipulators". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1036 –1041 (cit. on p. 23).

Pratt, Jerry, Chee-Meng Chew, Ann Torres, Peter Dilworth, and Gill Pratt (2001). "Virtual Model Control: An Intuitive Approach for Bipedal Locomotion". In: *The International Journal of Robotics Research* 20.2, pp. 129–143 (cit. on p. 7).

Quigley, Morgan, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng (2009). "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software* (cit. on pp. 9, 12, 14, 25).

Reckhaus, Michael, Nico Hochgeschwender, Paul G. Ploeger, and Gerhard K. Kraetzschmar (2010). "A Platform-independent Programming Environment for Robot Control". In: *1st International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob)* (cit. on p. 21).

Schlegel, Christian, Thomas Haßler, Alex Lotz, and Andreas Steck (July 2009). "Robotic software systems: From code-driven to model-driven designs". In: *International Conference on Advanced Robotics (ICAR)*, pp. 1–8 (cit. on p. 21).

Schultz, Ulrik P., David Christensen, and Kasper Støy (2007). "A Domain-Specific Language for Programming Self-Reconfigurable Robots". In: *Automatic Program Generation for Embedded Systems (APGES)* (cit. on pp. 22, 96).

Selig, Jon M. (2005). *Geometric Fundamentals of Robotics*. Ed. by David Gries and Fred B. Schneider. Springer (cit. on p. 56).

Semini, Claudio, Jonas Buchli, Marco Frigerio, Thiago Boaventura, Michele Focchi, Emanuele Guglielmino, Ferdinando Cannella, Nikos G. Tsagarakis, and Darwin G. Caldwell (Apr. 2011). "HyQ – A Dynamic Locomotion Research Platform". In: *International Workshop on Bio-Inspired Robots* (cit. on p. 15).

Semini, Claudio, Hamza Khan, Marco Frigerio, Thiago Boaventura, Michele Focchi, Jonas Buchli, and Darwin G. Caldwell (2012). "Design and Scaling of Versatile Quadruped Robots". In: *Climbing and Walking Robots (CLAWAR)* (cit. on p. 81).

Semini, Claudio, Nikos G. Tsagarakis, Emanuele Guglielmino, Michele Focchi, Ferdinando Cannella, and Darwin G. Caldwell (2011). "Design of HyQ – a Hydraulically and Electrically Actuated Quadruped Robot". In: *IMechE Part I: J. of Systems and Control Engineering* 225, pp. 831–849 (cit. on pp. 15, 64, 92).

Sentis, Luis and Oussama Khatib (2005). "Synthesis of whole-body behaviors through hierarchical control of behavioral primitives". In: *International Journal of Humanoid Robotics* 2.4, pp. 505–518 (cit. on pp. 11, 18, 48).

Sherman, Michael and Dan Rosenthal (2013). *SD/FAST*. URL: www.sdfast.com/ (cit. on pp. 14, 24).

Siciliano, Bruno, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo (2009). *Robotics. Modelling, Planning and Control*. Ed. by Michael J. Grimble and Michael A. Johnson. Springer (cit. on pp. 18, 45, 46, 49, 50, 58).

Schaal, Stefan (2009). *The SL simulation and real-time control software package*. Tech. rep. CLMC lab, University of Southern California (cit. on pp. 25, 41, 84, 88).

Smith, Russell (2013). *The Open Dynamics Engine simulation library*. URL: www.ode.org (cit. on p. 3).

Snyder, Alan (June 1986). "Encapsulation and inheritance in object-oriented programming languages". In: *ACM SIGPLAN Notices* 21.11, pp. 38–45. ISSN: 0362-1340 (cit. on p. 36).

Steck, Andreas and Christian Schlegel (Sept. 2010). "Towards Quality of Service and Resource Aware Robotic Systems through Model-Driven Software Development". In: *1st International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob)* (cit. on p. 21).

Stuelpnagel, John (Oct. 1964). "On the Parametrization of the Three-Dimensional Rotation Group". In: *SIAM Review* 6.4, pp. 422–430 (cit. on p. 50).

Toogood, R. W. (May 1989). "Efficient robot inverse and direct dynamics algorithms using microcomputer based symbolic generation". In: *IEEE International Conference on Robotics and Automation*, 1827 –1832 vol.3 (cit. on p. 23).

Tsagarakis, Nikos G., Stephen Morfey, Gustavo Medrano Cerda, Zhibin Li, and Darwin G. Caldwell (2013). "Development of Compliant Humanoid robot COMAN: Body design and Stiffness Tuning". In: *IEEE International Conference on Robotics and Automation (ICRA)*. [accepted for publication] (cit. on pp. 16, 17, 51, 94).

Yaghmour, Karim, Jon Masters, Gilad Ben-Yossef, and Philippe Gerum (2008). *Building Embedded Linux Systems*. Ed. by Andy Oram. O'Reilly (cit. on p. 32).